COUPLING INTERFACE FOR PHYSICS-TO-SYSTEM SIMULATIONS

A Thesis

by

MICHAEL LEE LEIMON

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Pavel Tsvetkov
Committee Members,	Graham Allen
	Karen Vierow
Department Head,	Yassin Hassan

December 2012

Major Subject: Nuclear Engineering

ABSTRACT

A new interfacial code was developed to couple the reactor physics code PARCS/AGREE to the systems level code MELCOR, with a goal of enabling state-of-art transient event analysis for high temperature gas reactor designs. Following the completion of this new code, it was then demonstrated by running two different coupled simulations, one of which was a transient event.

The resultant code is capable of coupling spatial power profiles, point kinetics information and transient reactivity values from PARCS/AGREE to MELCOR by means of input/output file manipulation. The coupling demonstrations were between PBMR400 models that were designed to have an equivalent core region nodalization to that which was used in the OECD/NEA PBMR400 benchmark, thus allowing for comparisons.

The accessible coupled simulation output results as extracted from MELCOR appeared to be overly generalized. Even so, the axial profiles from the coupled steady-state demonstration were in good agreement with the axial profiles of other OECD/NEA participants. Conversely, the coupled transient simulations showed a suspect, maximum average nodal component temperature rise of approximately 0.4K from a 3+$ reactivity insertion.

DEDICATION

This thesis is dedicated to my family: Sara, John, Tom and Carol Leimon.

ACKNOWLEDGEMENTS

There are a number of people and groups which I absolutely must recognize for helping me get this thesis to fruition. First and foremost, I must thank my advisor, Dr. Tsvetkov, for all of his hard work, support, and excellent guidance. Next, I must thank my committee members Dr. Vierow and Dr. Allen for their assistance with the thesis and the defense. Another individual who I must thank is Brad Beeny. Brad helped me considerably with the development and testing of the MELCOR portion of the coupling code.

I must also show my gratitude to the people of the PARCS/AGREE development team, especially Volkan Seeker and Yunlin Xu. Similarly, I owe thanks to the entire MELCOR development team, especially Mike Young. For all their support, feedback, patience and for their sense of humor, I must thank all my fellow members of Dr. Tsvetkov's Advanced Energy Technologies Research Group. Also, I need to thank all staff members of the TAMU Nuclear Engineering department, especially Jennifer House, whose help with innumerable things made my defense possible. To both the members of the breakfast club and the Gumby's crew: Vishal, Gentry, Sara, Matt, Sangjoon, Ryan, Austin, Sandeep, et al; I thank you for giving me days to look forward to in the midst of every week.

Last but not least, I give my sincere gratitude to the NRC, which was responsible for the funding of this research.

NOMENCLATURE

AGREE	Advance Gas REactor Evaluator
DOE	U.S. Department of Energy
EPACT05	Energy Policy Act of 2005
FPT	Fission Particle Transport
GIF	Generation IV International Forum
HTGR	High Temperature Gas Reactor
LWR	Light Water Reactor
NEA	Nuclear Energy Agency
NGNP	Next Generation Nuclear Plant
NRC	Nuclear Regulatory Comission
OECD	Organization for Economic Cooperation and Development
PARCS	Purdue Advanced Reactor Core Simulator
PBMR	Pebble-Bed Modular Reactor
PMR	Prismatic-block Reactor
RELAP	Reactor Excursion and Leak Analysis Program
TRAC	Transient Reactor Analysis Code
TRACE	TRAC/RELAP Advanced Computational Engine
VHTR	Very High Temperature Reactor

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

An interface code was be developed to couple a reactor physics code to a systems level code with the goal of enabling state-of-the-art transient analysis capability for high temperature gas-cooled reactor (HTGR) designs. This particular implementation coupled the systems level code MELCOR with the reactor physics code PARCS/AGREE. The need for such a coupling arose from the particular difficulties of modeling HTGR designs.

A common use of MELCOR is to model the progression of severe accidents. Among its many features, it is capable of modeling fission particle transport (FPT), which is very important for calculating radioactive product release during accident scenarios. By coupling information from PARCS/AGREE, this code which was originally designed for use with light water reactor (LWR) designs, can be used on HTGR designs.

1.1 Background

In 2001, fueled by the desire to advance nuclear energy technology, a number of countries chartered an international forum to consolidate research efforts [2]. There are in total eight goals for the Generation IV reactors, which fall into the following four categories: sustainability, economics, safety and reliability, and proliferation resistance. Sustainability goals are to provide long-term energy generation that meets clean air objectives while increasing fuel utilization, as well as to minimize production of nuclear waste and the burden of its storage. Economic goals include, having a cost advantage over competing energy sources as well as reducing the financial risk associated with nuclear energy. Safety and Reliability goals are: having a low chance of core damage, elimination of need for offsite emergency response, and in

general excellent reliability and safety. The proliferation resistance goal is to make Generation IV reactors the least desirable source for weapons-material diversion [5].

The Generation IV International Forum (GIF), which now has 13 member countries, selected six promising reactor systems for further study. Among these reactor systems was very high temperature reactor (VHTR) system, which had a number of different core designs including the pebble bed modular reactor (PBMR) as well as the prismatic block reactor (PMR). It is worth noting that both the PBMR and PMR designs are classified as HTGR designs as well.

On July 29, 2005, the United States Congress passed a bill named the Energy Policy Act of 2005 (EPACT05), which was soon after signed into law by then President George W. Bush. The overarching goal of this act was to combat growing energy problems in the country. Title VI, Subtitle C of the EPACT05, established the Next Generation Nuclear Plant (NGNP) project. In this outline, a nuclear plant should be developed which could be used for both energy and hydrogen production. Section 644 of the EPACT05 describes the involvement of the Nuclear Regulatory Commission (NRC) with the NGNP. More specifically, SEC.644(b)(2) of the EPACT05, describes a required part of the NRC licensing strategy for the NGNP, "a description of analytical tools that the Nuclear Regulatory Commission will have to develop to independently verify designs and performance characteristics of components, equipment, systems, or structures associated with the prototype nuclear reactor;" [3].

To satisfy provisions laid out in the EPACT05, the NRC and the U.S. Department of Energy (DOE) jointly submitted a licensing strategy for the NGNP in a report to congress in August 2008. It is in this report that the DOE mentioned that it had chosen the HTGR design for the NGNP. Due to some unique differences between a LWR and HTGR designs, certain regulatory requirements for LWRs were not applicable to HTGRs. As such, the NRC staff needed to adapt the LWR regulatory requirements

for HTGRs. Additionally, the NRC must develop analytical tools capable of analyzing among other things: accidents, high temperature materials, fuel performance, and fission product transport for the HTGRs. The development strategy provided by NRC was to use existing analytical tools with modifications to make them better suited for the this purpose. The analytical tools mentioned specifically by the NRC in their development strategy for the NGNP were, MELCOR and PARCS [19].

1.2 HTGR Features

The modeling of HTGR designs is complicated by a number of reactor physics features including:

- Neutron streaming effects along helium coolant channels,
- upper cavity streaming,
- spatial power oscillations of tall, thin annular cores,
- fuel double heterogeneity,
- buckling feedback in the spectrum,
- graphite scattering kernel [30].

Another aspect of HTGR designs which complicates modeling are related to thermohydraulics, more specifically:

- local thermal non equilibrium (LTNE),
- porous medium heat transfer [24],
- packed bed flow and pressure drop (PBMR only) [23].

Additionally, since a goal of generation IV reactors is improved efficiency, these designs will likely experience longer irradiation periods, thus requiring more precise nuclear data for transuranic actinides. Another complication regarding the PBMR design is addressing the issue of fuel pebble movement.

1.3 Thesis Objectives

The objective of this thesis is to create an interface to couple the reactor physics code PARCS/AGREE to the systems level code MELCOR. This research should:

1. Develop coupling algorithms:
    - coupling of spatial power distribution,
    - coupling of point kinetics data.
2. Implement an interface,
    - handle translation and information passing between codes,
    - automate the launching of computational codes,
    - inclusion of rudimentary output plotting functionality.
3. Verify algorithm function.
    - demonstrate a steady state coupling of a PBMR400 model,
    - demonstrate a transient coupling of a PBMR400 model.

1.4 Implementation

1.4.1 Data Coupling

Spatial power information from PARCS is loaded as power density from the '.map' output files. The PARCS input file corresponding to that particular output

file is then parsed to locate the thermohydraulic nodalization which corresponds to these power density values. From this nodalization, the volumes of each node will be calculated and stored for use when converting to a form usable by MELCOR. Multiplying the power density of the thermohydraulic nodes with their respective volumes resulted in power per node. In the case of specifying power by using control function heat sources, these power per node values will be used directly and mapped to equivalent nodes. In the case of specifying relative power densities, these power per node values are used to determine the relative power produced by each axial section and radial ring.

The point kinetics information generated by PARCS is extracted from the '.pkd' output file. This information is supplied to MELCOR by utilizing a couple of different methods: using COR sensitivity coefficients input records (COR_SC) and by using an external data file (EDF) in conjunction with a control function (CF) to supply the reactivity value as a function of time.

In the PARCS point kinetics output, a single fuel density feedback reactivity coefficient and a single moderator density feedback reactivity coefficient are supplied. However MELCOR uses a second order polynomial for fuel density feedback reactivity and a fourth order polynomial for graphite density feedback reactivity [4]. So to make these equivalent, aside from setting the first order coefficients of both, the second and second through fourth order coefficients for fuel and graphite density feedback reactivity respectively, are also set to zero in the resulting MELCOR input. In the current release of MELCOR there is no model for handling xenon, which restricts its usage to short time scale transients. Also, MELCOR does not differentiate between graphite that is part of the fuel pellets from that of the central and outer reflectors. This is important because they have distinctly different temperature reactivity coefficients. However it is possible to supply the decay constants as well as

the relative abundances of the delayed neutron groups.

The process of supplying time dependent reactivity information is done by utilizing control functions which result in reading values from an external data file. To accomplish such a coupling, there must first be a translation between the time at which the transient incident occurs in the PARCS model simulation time to that of the MELCOR model simulation time. Next these time adjusted reactivity values must be populated into an EDF. To complete the process and allow this information to be used, a number of control functions must be created to: understand the structure of the EDF, read information from the EDF, trigger the start of the transient, and to trigger the end of the transient. Point kinetics information generated by PARCS is outputted to a file. This information includes the following reactivity coefficients: fuel temperature, moderator temperature, xenon concentration, and reflector temperature. Also produced by PARCS are the following values as a function of time: total reactivity, generation time, pcfact, plevel and betas for six groups. In the case of a transient problem, it is important that this point kinetics information is transferred to MELCOR.

1.4.2 Coupling Interface

The coupling code was written using the Python 3 programming language. This allows for cross platform operation of the code. Currently support is planned for Windows and partial support for both Linux and OSX. The transfer of information from PARCS/AGREE to MELCOR is accomplished via input file exchange. Also, it is possible to create a section of an input file and either splice it in, or replace an equivalent section in existing MELCOR input file.

The structure of the coupling code is in the form of several modules. These modules will expand on the capabilities of the base coupling interface by providing

additional functions, native datatypes and their methods. Additionally, the coupling algorithms for PARCS/AGREE and MELCOR reside within these separate modules. One advantage of this structure is, code isolation within the modules, thereby reducing the chance of code regression when either adding new modules or adding extra functionality elsewhere.

One of the goals of the project is to support the various HTGR designs. As such, the coupling methods that were developed should be general enough that their usage may be applicable to these various reactor designs. Furthermore, there should be consistency in the usage of these methods (e.g. the coupling process for a PBMR should be roughly the same as for a PMR) to reduce the learning curve for the interface. However, the interface demonstration included with this thesis will only be covering the PBMR400 design.

Other design goals for the interface are: inclusion of an advanced error reporting system, a simplified input style, and inclusion of graphical post-processing techniques. The error reporting system is capable of reporting the nature of an error, echoing the raw input of the erroneous line, as well as reporting the line number of the erroneous line. Another desirable feature is variable declaration, which would simplify input reuse and allow for resulting input files to be both more concise and less redundant. It should also be possible to generate plots of some information from either PARCS/AGREE or MELCOR, by an automated process of gnuplot input generation and subsequent execution.

2. MATHEMATICAL PROBLEM FORMULATION

The point kinetics equations are capable of describing transient events under the assumption that the shape of the neutron flux remains constant. As such, one method to effectively couple a transient event from one code to another, is by passing along point kinetics information. Presented in this chapter is a general formulation of these equations as well as the specific formulations used by PARCS and MELCOR exactly as described by their respective theory manuals. Both the PARCS and MELCOR formulations for the equations must be equivalent to enable this sort of transient coupling.

2.1 A General Point Reactor Kinetics Formulation

For the sake of simplicity, one general form of the point kinetics equations is presented here:

$$\begin{aligned} \frac{dn(t)}{dt} &= \frac{\rho(t)-\beta}{\Lambda} n(t) + \sum_{i=1}^{N} \lambda_i C_i(t), \\ \frac{dC_i(t)}{dt} &= \frac{\beta_i}{\Lambda} n(t) - \lambda_i C_i(t), \\ & \qquad i = 1, ..., N. \end{aligned} \tag{2.1}$$

where,

$n(t)$ = amplitude function,

$\rho(t)$ = reactivity,

$\beta(t)$ = total delayed neutron fraction,

Λ = neutron generation time,

λ_i = decay constant for effective delayed group i,

$C_i(t)$ = effective precursor density,

β_i = effective delayed neutron fraction of delayed neutron group I,

This form of the equations is adapted from the referenced version, which specified a variant utilizing exactly six delayed neutron groups, without an external neutron source, $Q(t)$ [26].

2.2 PARCS Formulation

The following formulation for the point kinetics equations, is provided in the exact manner as it was presented in the PARCS theory manual. This is done in effort to highlight the non-triviality of such an equivocation.

PARCS uses the following general form of the point kinetics equations,

$$\begin{aligned} \frac{dp(t)}{dt} &= \frac{\rho(t) - \beta^{eff}(t)}{\Lambda(t)} p(t) + \frac{1}{\Lambda_0} \sum_k \lambda_k(t)\zeta(t) \\ \frac{d\zeta_k(t)}{dt} &= \frac{\Lambda_0}{\Lambda(t)} \beta_k^{eff}(t)p(t) - \lambda_k(t)\zeta(t), \\ & \qquad k = 1, 2, ..., N_d. \end{aligned} \tag{2.2}$$

where the neutron mean generation time $\Lambda(t)$ is defined as:

$$\Lambda(t) = \frac{\left\langle \phi_g^*(,) \frac{1}{\nu_g(,)} \psi_g(,t) \right\rangle}{F(,t)}, \tag{2.3}$$

and the reduced precursor concentration is:

$$\zeta_k(t) = \frac{\left\langle \phi_g^*()\chi_{dk,g}()C_k(,t)\right\rangle}{F_0}, \tag{2.4}$$

and where,

$$\lambda_k(t) = \frac{\left\langle \phi_g^*()\chi_{dk,g}()C_k(,t)\right\rangle}{\zeta_k(t)F_0}, \tag{2.5}$$

$$\beta^{eff}(t) = \sum_k \beta_k^{eff}(t), \tag{2.6}$$

$$\beta_k^{eff}(t) = \frac{\left\langle \phi_g^*()\chi_{dk,g}()\beta_k()\hat{S}^F(,t)\right\rangle}{F(t)}, \tag{2.7}$$

$$\rho(t) = \frac{1}{F(t)}\left\langle \phi_g^*()(M_g - F_g)\psi(,t)\right\rangle, \tag{2.8}$$

$$F(t) = \left\langle \phi_g^*()\chi_g()\hat{S}^F(,t)\right\rangle, \tag{2.9}$$

$$\hat{S}^F(,t) = \frac{S^F(,t)}{p(t)} = \frac{1}{k_{eff}^s}\sum_{g'} \nu\Sigma_{f,g'}(,t)\psi_{g'}(,t), \tag{2.10}$$

$$M_g[\psi(,t)] = \nabla\cdot(D_g\nabla\psi(,t)) + \sum_{g'}\Sigma_{g,g'}(,t)\psi_{g'}(,t), \tag{2.11}$$

$$F_g[\psi(,t)] = \chi_g()\hat{S}^F(,t). \tag{2.12}$$

[10]

2.3 MELCOR Formulation

The following formulation for the point kinetics equations, is provided in the exact manner as it was presented in the MELCOR theory manual. As was stated previously, this is done in effort to highlight the non-triviality of such an equivocation.

MELCOR uses a six-delayed group variant of the point kinetics equations,

$$\frac{dn}{dt} = \frac{\rho - \beta}{\Lambda} + \sum_{i=1}^{6} \lambda_i C_i + S_0$$
$$\frac{dC_i}{dt} = \frac{\beta_i}{\Lambda} n - \lambda_i C_i \tag{2.13}$$

where,

n = prompt neutron power [W],

ρ = reactivity,

β = total delayed neutron fraction,

Λ = prompt neutron generation time [s],

λ_i = decay constant for effective delayed group i [s^{-1}],

C_i = power of i^{th} delayed precursor group [W],

β_i = fraction of i^{th} delayed neutron group,

S_0 = initial neutron source [W/s].

The value for reactivity (ρ) is defined as:

$$\rho = \rho_{ext} + \rho_D + \rho_f + \rho_G, \tag{2.14}$$

where,

ρ_{ext} = external reactivity insertion,

ρ_D = Doppler feedback,

ρ_f = fuel thermal density feedback,

ρ_G = graphite thermal density feedback. [22]

Examining the reactivity components ρ_D, ρ_f and ρ_G in more detail:

The fuel Doppler reactivity is modeled using the following logrithmic function,

$$\rho_D = \chi_D ln\left(\frac{\overline{T_f}}{T_f^0}\right) \tag{2.15}$$

where,

χ_D = Doppler coefficient,

$\overline{T}_f$ = average fuel temperature [K],

T_f^0 = volume averaged fuel temperature at initial steady state reference [K].

The model for fuel density feedback reactivity is a second order polynomial,

$$\rho_f = \chi_{f,1}\left(\overline{T}_f - T_f^0\right) + \chi_{f,2}\left[\left(\overline{T}_f\right)^2 - \left(T_f^0\right)^2\right], \tag{2.16}$$

where,

$\chi_{f,1}$ = fuel expansion temperature reactivity coefficient [K^{-1}],

$\chi_{f,2}$ = fuel expansion temperature reactivity coefficient [K^{-2}],

$\overline{T}_f$ = average fuel temperature [K],

T_f^0 = volume averaged fuel temperature at initial steady state reference [K].

The model for graphite density feedback reactivity is a fourth order polynomial,

$$\rho_G = \sum_{m=1}^{4} \chi_{G,m}\left[\left(\overline{T}_G\right)^m - \left(T_G^0\right)^m\right], \tag{2.17}$$

where,

$\chi_{G,m}$ = graphite expansion temperature reactivity coefficient [K^{-m}],

$\overline{T}_G$ = average graphite temperature [K],

T_G^0 = volume averaged graphite temperature at initial steady state reference [K].

The default values for the reactivity coefficients were obtained from a least-squared fit of calculations performed by INEEL and then reported by MacDonald in 2003 for the NGNP prismatic reactor. These default values appear as such:

$$\chi_D = -0.022, \quad \chi_{f,1} = -4.78 \times 10^{-5}\,[K^{-1}], \quad \chi_{f,2} = -6.75 \times 10^{-9}\,[K^{-2}],$$
$$\chi_{G,1} = +14.834 \times 10^{-5}\,[K^{-1}], \quad \chi_{G,2} = -1.6025 \times 10^{-7}\,[K^{-2}],$$
$$\chi_{G,3} = +6.9907 \times 10^{-11}\,[K^{-3}], \quad \chi_{G,4} = -1.1142 \times 10^{-14}\,[K^{-4}]. \quad [21]$$

2.4 Reconciled Differences

After working with the output produced by PARCS, a number of reconcilable differences appeared. The value for fuel expansion temperature reactivity used by MELCOR comes from a second order polynomial. Since only a single fuel expansion temperature reactivity coefficient appears in the PARCS output, the value for the second coefficient in MELCOR is set to zero, thus equivocating the resulting value. Similarly, the value for graphite expansion temperature reactivity used by MELCOR comes from a fourth order polynomial. Since only a single graphite expansion temperature reactivity coefficient appears in the PARCS output, the value for the second through fourth coefficients in MELCOR are set to zero, thus equivocating the resulting value.

3. REACTOR FEATURES AND SYSTEM-LEVEL SIMULATION NEEDS

3.1 Complicating Reactor Physics Features

There are a number of reactor physics features of HTGR designs which result in modeling complications. Some of these more notable reactor physics features are: neutron streaming, spatial power oscillations, fuel double heterogeneity, buckling feedback in the spectrum, and graphite scattering kernels.

3.1.1 Neutron Streaming

For the case of neutron diffusion through media containing cavities, there can be a complication if these cavities are not small (compared to the mean free path in the media) and homogeneously distributed. For the situation described here, simple homogenization of the system will not produce a good approximation. In this situation, the mean-square-free-path will be strongly affected by the shape of the cavities in the media. This effect is often referred to as neutron streaming and it has an effect on neutron leakage and subsequently on core criticality as well [18].

Due to the design of the pebble bed reactors, they inherently contain an inhomogenously distributed number of irregularly sized and shaped cavities within the core media. As such, these reactor designs require special treatment to account for the effect of neutron streaming. Moreover, this effect must be accounted for both within the media itself as well as for the unpopulated volume above the fuel, upper cavity streaming.

3.1.2 Spatial Power Oscillations

Due to the tall-thin annular design of the HTGR cores, they are susceptible to spatial power oscillations. These spatial power oscillations are induced by localized

changes in Xenon concentration, which may occur naturally or as a result of control rod movement. For spatial power oscillations to be possible, the core design must be large enough that parts of the core are spatially de-coupled from other parts. In most cases, this requirement is satisfied by having one dimension of the core (either radial or axial) be several times larger than the neutron diffusion length in that material. For the case of graphite, the neutron diffusion length is around 59cm. Since most HTGR designs tend to be greater than 8m in height, they are most certainly susceptible to spatial power oscillations in the axial dimension [27].

3.1.3 Fuel Double Heterogeneity

The fuel of choice for modern HTGR designs utilizes TRISO particles. TRISO particles are small spherical particles which contain fuel at their center and the following outward encompassing layers in this particular order: C, PyC, SiC, and PyC. The resultant fuel for modern HTGR designs has many TRISO fuel particles embedded into a graphite matrix which is then surrounded by a graphite layer and formed into the appropriate shape (spheres or right circular cylinders). Due to the tight and random packing of fuel particles next to one another, the neutron interactions and their effective slowing due to fuel, are complex and can not be ignored [14].

3.1.4 Buckling Feedback in the Spectrum

Due to the higher operating temperatures of HTGRs, their neutron spectra are more susceptible to forms of buckling feedback. Instantaneous changes in flux levels will result in corresponding instantaneous changes in buckling, or leakage. These changes of leakage will cause a feedback effect on the resultant neutron spectrum. The resulting neutron spectrum will be changed since the buckling feedback for fast neutrons differs from that of thermal neutrons [29].

3.1.5 Graphite Scattering Kernel

In general, a scattering kernel is the density of a probability that an incident particle of a given velocity is scattered into a different specified velocity upon contact with a medium [8]. Since graphite is used as the moderator in HTGR designs, understanding its scattering kernel is critical to correctly predicting the rate of energy loss by neutrons. Since graphite is a layer structure and not an isotropic lattice; the more common lattice vibration models can not be used to accurately describe neutron scattering within graphite [28].

3.2 Themal-hydraulic Features

Another source of modeling complications for HTGR designs is due to their thermal-hydraulic features. A number of these important thermal-hydraulic features deserve more explanation, namely: local thermal nonequilibrium (LTNE), porous medium heat transfer, as well as packed bed flow and pressure drop.

3.2.1 Local Thermal Nonequilibrium

When considering the situation of a porous matrix with fluid flowing through it; if the matrix and fluid happen to be at two different temperatures locally, then this situation is a case of local thermal nonequilibrium. Local thermal nonequilibrium is significant in that it requires the usage of more complex formulation to analyze correctly, than for cases of local thermal equilibrium [16]. Also, this phenomenon is applicable for PBMRs, where there is helium flowing through a packed bed of graphite pebbles.

3.2.2 Porous Medium Heat Transfer

The idea of porous medium heat transfer is based upon the concept of porous media. More specifically, porous media is defined as material volume containing a

solid matrix that is otherwise filled with void. Porous materials allow fluid flow though, however the fluid will experience complex changes in flow velocities and pressure drop though the material. Porous materials may be characterized by the following parameters: porosity, permeability, and tortuosity [17].

3.2.3 Packed Bed Flow

Another thermal-hydraulic factor which complicates analysis of PBMR designs is the analysis of fluid flow through a packed bed of randomly distributed spheres. Heat transfer capability is highly variable for situations of flow around curved surfaces. Moreover, the heat transfer capability is highly dependent on the size of the gaps between the spheres as well as the radius of curvature of the spheres. To properly simulate all of the flow effects, a very fine computational mesh must be used in a fluid dynamics simulation [15].

4. PHYSICS AND SYSTEM CODES AND THEIR ROLES IN THE INTERFACE

4.1 PARCS/AGREE Simulation Codes

4.1.1 PARCS Code

The original NRC version of PARCS was released back in 1998. Since that original release, the code base has been updated considerably to using Fortran 90 (PARCS version 3.0 and later). PARCS is a three dimensional reactor core simulator which is capable of the following major features:

- eigenvalue calculations
- transient calculations
- Xenon transient calculations
- decay heat calculations
- pin power calculations
- depletion calculations
- adjoint calculations

Varying degrees of interoperability with PARCS has been established with the following codes [11]:

- TRACE

  The "TRAC/RELAP Advanced Computational Engine" is a best-estimate reactor systems code that is used to analyze neutronic-thermal-hydraulic behav-

ior of reactor systems during both steady-state and transient events [6]. This code is directly coupled to PARCS.

- RELAP5

  RELAP5 is a best-estimate code for simulating the coolant systems of light-water reactors during anticipated accident situations. This code simulates the thermal-hydraulic models of the primary and secondary systems, as well as the actions of the the reactor control system [12]. This code is coupled to PARCS using the PVM interface system.

- GENPMAXS

  The "Generation of the Purdue Macroscopic XS set" program was designed to operate as an interface between lattice physics codes such as: TRITON, HELIOS, and CASMO. This interoperability is achieved by the generation of PMAXS files from the results of the aforementioned lattice physics codes [31].

- TRITON

  TRITON is a control module of the modeling and simulation suite, SCALE. This particular control module is a two-dimensional depletion sequence [9]. Resultant cross section information is coupled to PARCS through the use of the GENPMAXS program.

- HELIOS

  This code is used for gamma flux and fuel burnup calculations. HELIOS is a two-dimensional transport program which allows flexible descriptions of system geometry [25]. Resultant cross section information is coupled to PARCS through the use of the GENPMAXS program.

- CASMO

CASMO is a lattice physics code that is developed primarily to analyze LWRs. The latest version, CASMO-5, is capable of performing two-dimensional transport calculations [20]. Resultant cross section information is coupled to PARCS through the use of the GENPMAXS program.

4.1.2 AGREE Code

AGREE is a thermal-fluids code, which acts as a driver for PARCS. Initially it was developed to model pebble bed reactors and so initially, it used a three dimensional porous medium approach with local thermal non-equilibrium. AGREE may be thought of as a completely rewritten and modern version of the THERMIX/DIRECKT codes.

To better support analysis of prismatic reactor designs, a couple of new features are in the process of being added to AGREE. The first of these new features is a new bypass flow model. The new bypass flow model allows for the addition of bypass channels to the fuel elements. These bypass channels allow for bypass flow as well as crossflow between fuel elements gaps resulting from irradiation damage. The other new feature being added to agree is a three dimensional triangular heat transfer model. This new heat transfer model will replace the r-θ-z model when used for prismatic reactor analysis. With the new model, the fuel element will be modeled as a hexagon which is then subdivided into six triangular regions or sectors. Each of these sectors will contain a coolant channel as well as a fuel compact. When used together, the bypass flow model and the new heat transfer model should result in an updated code capable of analyzing prismatic reactor designs [7].

4.2 MELCOR Simulation Code

MELCOR is developed by Sandia National Laboratories for the NRC. It is a fully integrated systems level code, that is used to model the progression of severe

accidents in LWRs. MELCOR is capable of treating the following phenomena:

- Thermal-hydraulic responses
  - of the reactor cavity
  - of reactor coolant systems
  - of containment
- Treatment of fission products
  - release
  - transport
- Treatment of Hydrogen
  - production
  - transport
  - combustion
- Core failure sequences
  - heat-up
  - degradation
  - relocation
- Core-concrete attack

Most notably, MELCOR is used to calculate the source term release from given accident scenarios [13].

4.2.1 Recent Updates to Support HTGR Designs

In version 2.1 of MELCOR, released in September 2008, there was notable addition of features to better support GCR designs. Two new reactor models were added, PBR for the pebble bed reactor and PMR for the prismatic reactor. These new models provided additional component types such as: pebble fuel, reflectors, and hexagonal graphite blocks. Other changes include, though not exhaustively: addition of cell to cell radiative/conductive heat transfer, modification of axial cell to cell conductivity, changes in coolant friction factor, and the addition of graphite oxidation models [32].

4.3 Role of the Interface

The accident analysis evaluation model for HTGRs proposed by the NRC was broken down into four focuses: normal operation, nuclear data preprocessing, fission products during normal operation, and transient analysis event sequences. A modified schematic of the proposed accident analysis evaluation model is included here under Figure 4.1.

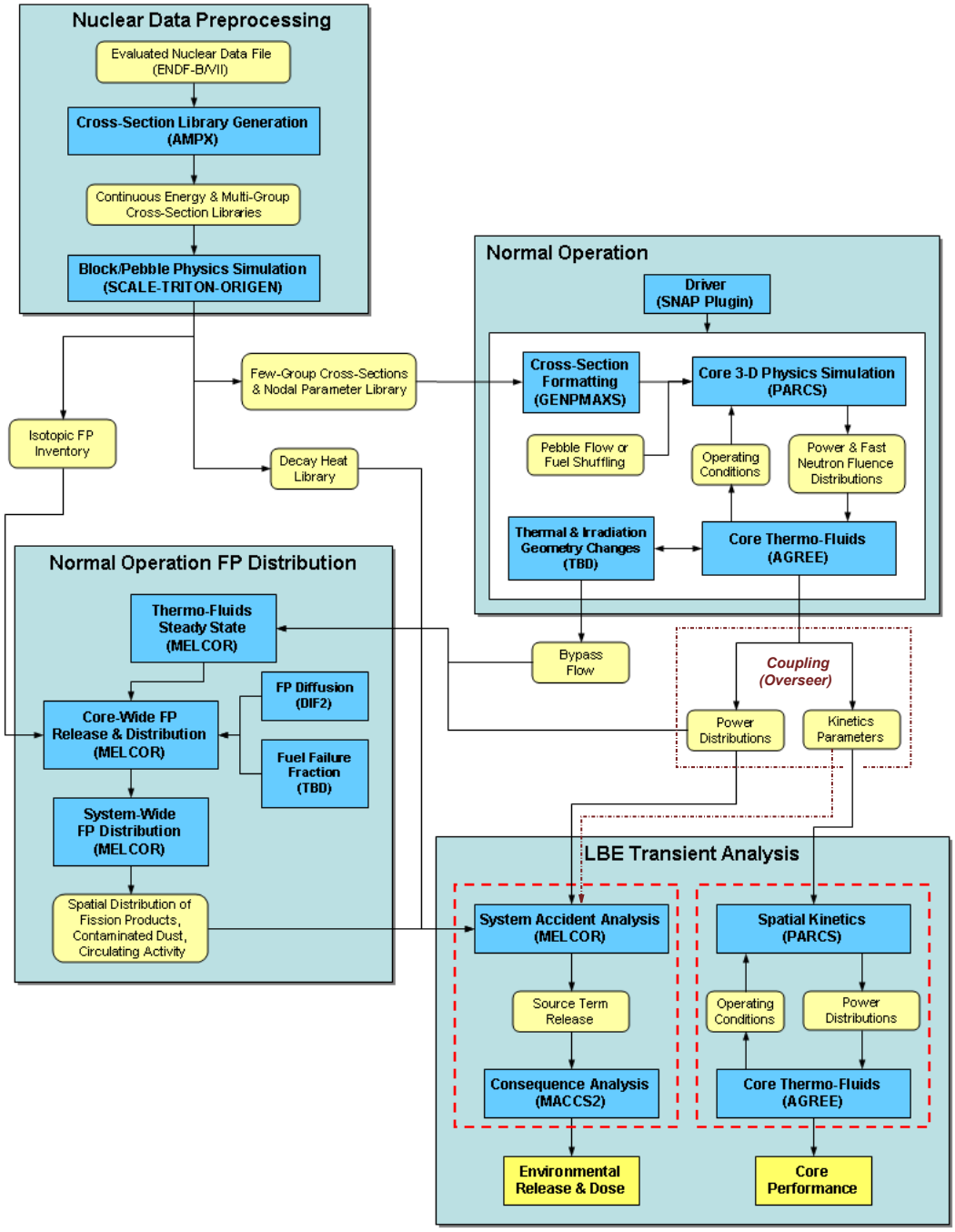

Figure 4.1: A modified schematic of the NRC accident analysis evaluation model concept for HTGRs [1]. The modifications (shown in maroon using dot-dash lines and italicized text) depict the functional role of the coupling code that was developed, Overseer.

4.3.1 Role of MELCOR

As can be seen in Figure 4.1, the overarching role of MELCOR is to determine the source term release from transient events. Of critical importance to this calculation is the fission product inventory of the reactor at the time of the accident. An accurate fission product inventory is to be calculated from the normal operation fission product focus of the NRC model. At the time of the accident, the MELCOR models for fission product transport will be used to calculate the resultant source term for an accident scenario.

4.3.2 Role of PARCS/AGREE

Both the PARCS and AGREE codes receive widespread use in the proposed NRC accident analysis evaluation model. The various roles of the codes are also depicted in Figure 4.1. In this model, PARCS is used to handle the neutronic aspects of the calculations. The AGREE code works closely with PARCS to address the various thermal-hydraulic aspects of the HTGR designs. Together the PARCS/AGREE codes provide accurate core power distributions as well as point kinetics parameters.

4.3.3 PARCS/AGREE-MELCOR coupling

In a one way coupling of the PARCS/AGREE code to MELCOR, the steady state power distributions produced by PARCS/AGREE must be transferred to MELCOR. This part of the coupling must be done so that an accurate fission product inventory for normal operations can be obtained. For the transient analysis, both power information as well as point kinetics parameters must be transferred from PARCS/AGREE to MELCOR so that the transient in MELCOR will occur exactly as the PARCS/AGREE codes describe. The result of this coupling should be an accurate source term release from an HTGR transient event.

5. COUPLING ALGORITHMS

The structure of the coupling code is in the form of several modules. These modules work by expanding on the capabilities of the base coupling interface by providing additional functions, native datatypes and their methods. The coupling methods developed were designed to be general enough that their usage may be applicable to various HTGR designs. This is accomplished by structuring the coupling interface as a large number of interconnecting micro-routines. The micro-routines are small and general in nature, functioning as building blocks from which more complex coupling routines for specific scenarios can be constructed. More specifically, these complex coupling routines can be constructed by the end-user, by the chaining of a number of micro-routines together.

Furthermore, there should be consistency in the usage of these methods (e.g. the coupling process for a PBMR should be roughly the same as for a PMR) to reduce the learning curve for the interface. However, the interface demonstration included with this thesis will only be covering the PBMR400 design.

Other design goals for the interface are: inclusion of an advanced error reporting system, a simplified input style, and inclusion of graphical post-processing techniques. The error reporting system should be capable of reporting the nature of an error, echoing the raw input of the erroneous line, as well as reporting the line number of the erroneous line. Another desirable feature is variable declaration, which would simplify input reuse and allow for resulting input files to be both more concise and less redundant. It should also be possible to generate plots of some information from either PARCS/AGREE or MELCOR, by an automated process of gnuplot input generation and subsequent exectuion.

5.1 Coupling Internals

The functions provided by the coupling interface and modules enable the user to utilize a number of different underlying algorithms. While the source code for the coupling interface is not included with this document, these underlying routines still deserve some amount of explanation. A number of these routines which find consistent use in coupling procedures are explored in the later portion of this section. The explanations presented in this section will in a general sense, describe how these underlying algorithms work.

5.1.1 The OVR Interpreter

One unique and complex aspect of the overseer interface is the ovr interpreter itself. This interpreter is responsible for managing all of the interactions between the user and the other algorithms provided to accomplish coupling procedures. To achieve the interface design goals of a simplified input style and inclusion of an advanced error reporting system, a significant amount of complexity was introduced into the ovr interpreter.

The algorithm depicted in figure 5.1 is a representation of the operational flow of the ovr interpreter. For any instance of using ovr, it is always in this algorithm that program execution starts. This diagram depicts cases of multiple possibilities as vertical lines joined by two or more connecting horizontal lines. The entry point for this diagram is from the top-left bubble.

When the coupling interface is started, it will check to see if any additional arguments were provided. There are three possible outcomes to this check: having no arguments will start an interactive shell, a valid argument will start the run_script routine, and finally an invalid argument will terminate ovr execution.

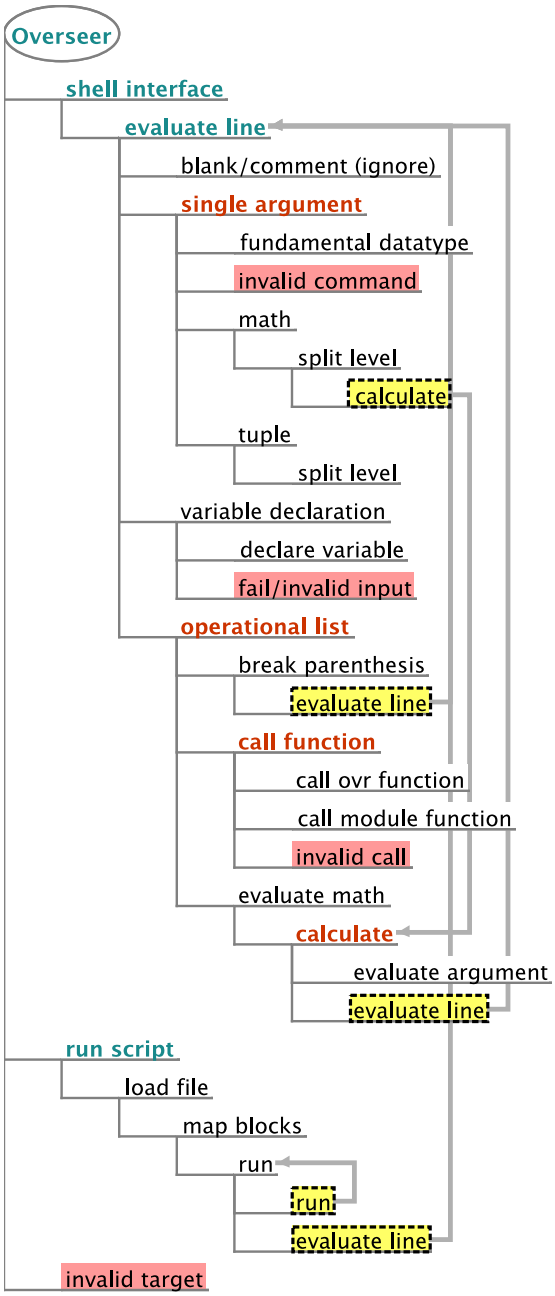

Figure 5.1: The program flow of the overseer coupling interface. Nodes with red backgrounds represent instances of failure and result in termination of the code execution (if in script mode). Nodes in dashed-boxes with a yellow background, call routines depicted elsewhere on the diagram.

One can see from the previous diagram, that the algorithm for the ovr interpreter is highly recursive. The nested linked recursion of both the *calculate* and *evaluate line*

sections of the code are required to handle the evaluation of variables, mathematical operations, and functions wherever they occur (e.g. a function may need to be evaluated within an element of a list that exists within an element of a list).

The process of line/substring evaluation by the interpreter is to first inspect the total number of whitespace separated items, then to categorize the type of evaluation. In each case, all of the whitespace separated elements are evaluated one by one (generally, from left to right). This evaluation process starts by determining whether or not an individual element is a fundamental datatype or something more complex. If an element is not a fundamental datatype, subsequent evaluations of the element continue until it is resolved into a fundamental datatype. Once all elements have been resolved to their most fundamental form, an evaluation is completed and the result is passed up a level. In this manner, the use of nested mathematical operations and function calls are enabled.

5.1.2 The parcs.run Function

The algorithm depicted in figure 5.2 is a representation of the operational progression of the function responsible for starting up PARCS/AGREE. This diagram depicts cases of multiple possibilities as vertical lines joined by two or more connecting horizontal lines. The entry point for this diagram is from the top-left bubble.

From this diagram it can be seen that, there are three possible failure modes for this function. All of these particular failure paths will result in the production of their own unique error message. Any error message produced by a function of an external module is captured and reported by the ovr interpreter after the completion of a toplevel *evaluate line* sequence (as notated in figure 5.1).

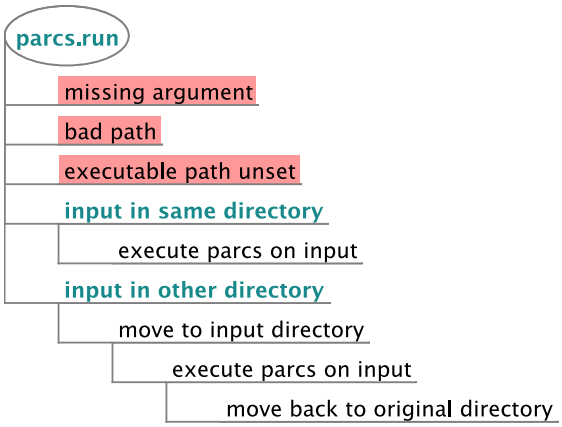

Figure 5.2: The program flow of the parcs.run routine. Nodes with red backgrounds represent instances of failure and result in termination of the code execution (if in script mode).

It can be seen from the preceding diagram, that the execution of the binary always takes place within the folder that contains the target input file, this is the desired behavior for a couple of reasons. At least one of the codes being coupled to in this research had an insufficient character limit for its target input argument. This character limit was such that, calling the executable with an absolute path could cause the program to fail (depending on where an input file was located). The second reason for this behavior is so that the output files for a particular program execution end up being created in the same folder as their corresponding input file.

5.1.3 The parcs.load_output Function

The algorithm depicted in figure 5.3 is a representation of the operational progression of the function responsible for loading output generated from PARCS/AGREE. This diagram depicts cases of multiple possibilities as vertical lines joined by two or more connecting horizontal lines. The entry point for this diagram is from the top-left bubble.

From this diagram it can be seen that, this operation proceeds for the most

part in a linear fashion. Essentially, this function will load all of the data files it is designed to handle, parse the files, and then store the information within a native parcs module datatype. Upon completion, this datatype will be returned to the ovr interpreter.

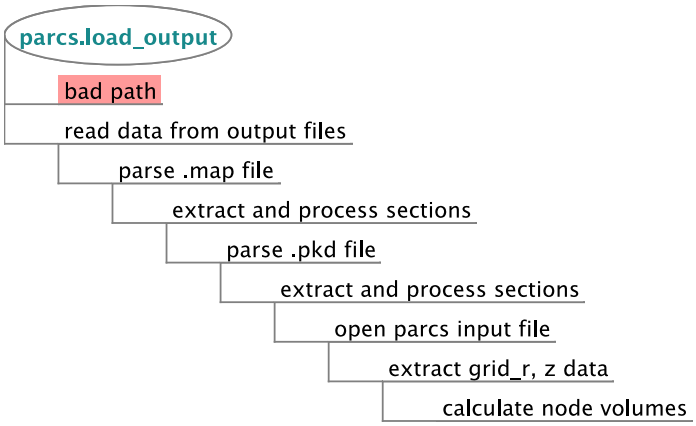

Figure 5.3: The program flow of the parcs.load_output routine. Nodes with red backgrounds represent instances of failure and result in termination of the code execution (if in script mode).

While it may initially seem odd that information from the specified PARCS input file is loaded by this routine, this is done for a very specific reason. The spatial power information reported in output files from PARCS is in the form of power density. Consequently, to specify spatial power to MELCOR, the total power of a control volume must be known. So it is from the PARCS input file that the spatial nodalization is read and subsequently used to calculate all of the volumes of the nodes. These calculated node volumes then allow for the conversion from PARCS power specification to a form acceptable for MELCOR.

5.1.4 The melcor.convert_power Function

The algorithm depicted in figure 5.4 is a representation of the operational progression of the function responsible for converting a PARCS native power datatype

into a MELCOR native power datatype. This diagram depicts cases of multiple possibilities as vertical lines joined by two or more connecting horizontal lines. The entry point for this diagram is from the top-left bubble.

From this diagram it can be seen that, there are a total of five different possible modes of failure. The first two failure cases will occur if the supplied input is blatantly incorrect. To determine if one of other three reportable errors exists, requires further inspection of the input and possibly of the datatype that the conversion is occurring from as well.

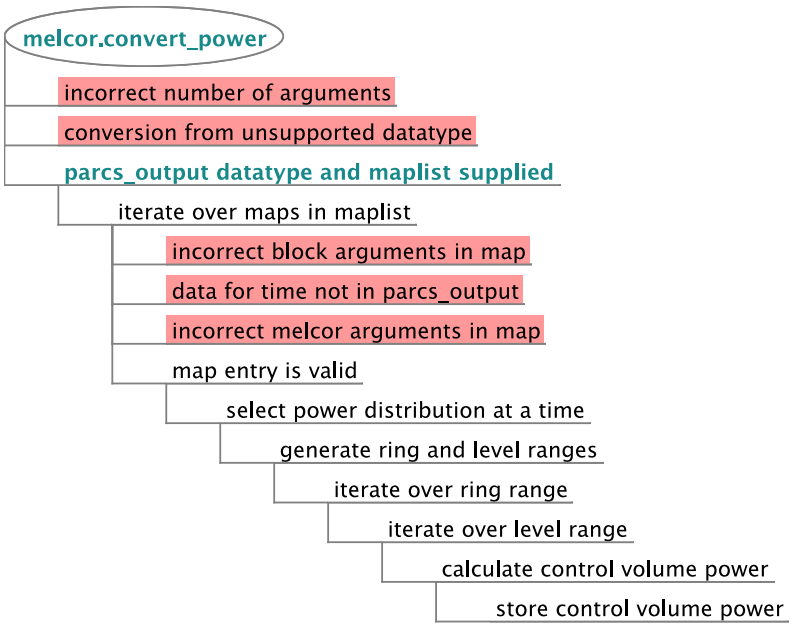

Figure 5.4: The program flow of the melcor.convert_power routine. Nodes with red backgrounds represent instances of failure and result in termination of the code execution (if in script mode).

Assuming there are no errors and execution proceeds, all of the information extracted will come from a single timestep. The spatial power density stored in a *parcs__power* datatype will be used in conjunction with the nodal volume information stored in the *parcs__output* datatype to generate the equivalent MELCOR control volume powers. Upon completion of this routine, a *melcor__power* datatype

is produced which can then be used for plotting or more importantly, for generating corresponding MELCOR input.

5.1.5 The melcor.generate_input Function: melcor__power Subroutine

The algorithm depicted in figure 5.5 is a representation of the operational progression of the function responsible for generating MELCOR input from a *melcor__power* datatype. This diagram depicts cases of multiple possibilities as vertical lines joined by two or more connecting horizontal lines. The entry point for this diagram is from the top-left bubble.

From this diagram it can be seen that, in actuality the melcor.generate_input function can utilize either a *melcor__power* datatype or a *melcor__pk* datatype. When used with a *melcor__power* datatype, there are two possible input generation styles: 'CF_SET' style and 'COR_ZP,RP' style. There are a number of technical differences between the two styles however, the output from this routine is similar in either case. This routine will return a string containing MELCOR input which represents the power information as specified in the MELCOR power datatype.

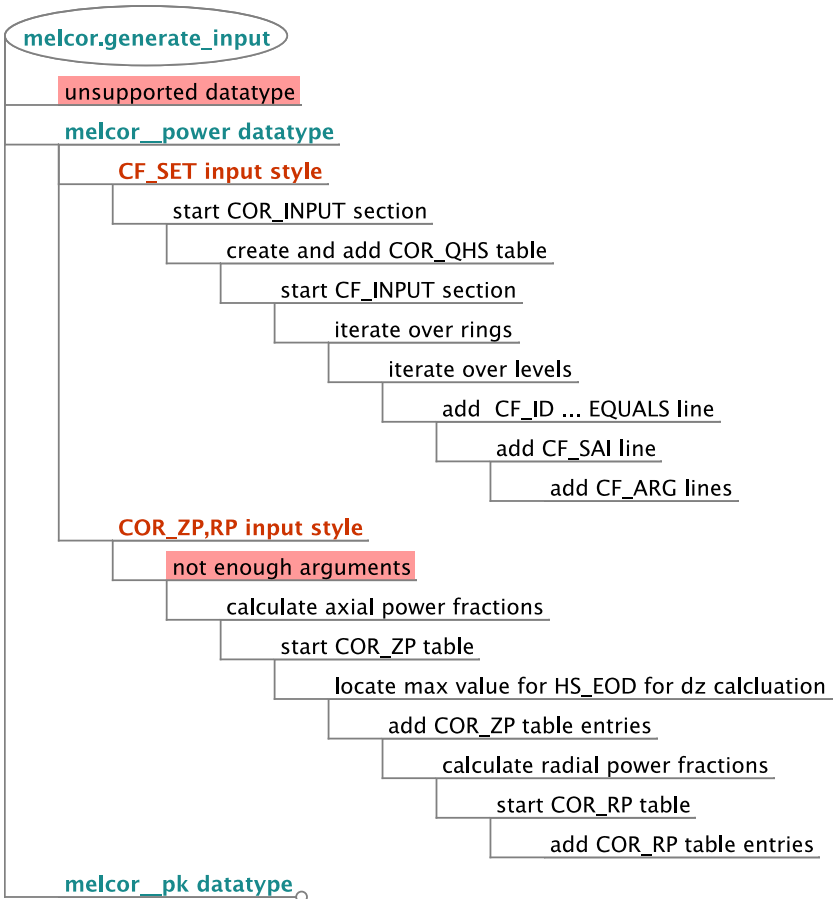

Figure 5.5: The program flow of the melcor.generate_input routine. This algorithm focuses on the generation of power input generation.

For more detailed information regarding the differences between the two input styles, please see section 6.6.1.4. This section covers from more of an end user standpoint, the usage of the melcor.generate_input function with the *melcor__power* datatype.

5.1.6 The melcor.generate_input Function: melcor__pk Subroutine

The algorithm depicted in figure 5.6 is a representation of the operational progression of the function responsible for generating MELCOR input from a *melcor__pk* datatype. This diagram depicts cases of multiple possibilities as vertical lines joined by two or more connecting horizontal lines. The entry point for this diagram is from

the top-left bubble.

From this diagram it can be seen that, in actuality the melcor.generate_input function can utilize either a *melcor__power* datatype or a *melcor__pk* datatype. When used with a *melcor__pk* datatype, there are three possible input generation styles: 'COR_SC' style, 'reactivity' style, and 'EDF' style. Unlike the multiple input styles used with the *melcor__power* datatype, the input styles for the *melcor__pk* datatype are not mutually exclusive. In fact, to describe a complex transient coupling it may be necessary to utilize all three input generation styles.

The 'COR_SC' input style generates MELCOR input of COR_SC sections 1404 and 1405. Section 1404 is the "Temperature Feedback Reactivity Coefficients" and section 1405 is the "Point Kinetics Model 6 Group Parameters" as presented in the MELCOR user manual.

For the COR_SC section 1404, parameters 2 (Fuel expansion temperature reactivity coefficient) and 4 (Graphite expansion temperature reactivity coefficient) are set with values extracted from PARCS output. However, since only the first order coefficients are returned by PARCS, the 2nd though 4th order coefficients must be set to zero to make the models equivalent. Due to bug with the version of MELCOR used to develop the coupling, the 4th order graphite expansion temperature reactivity coefficient could not be set to zero. For some reason, attempting to set this value in the input would cause MELCOR to crash.

For the COR_SC section 1405, all of the parameters (decay constants and relative abundances for delayed neutron groups 1-6) are set with values extracted from PARCS output. The value for the relative abundances is calculated from a set of β values taken from the first timestep of the *parcs__output* datatype that the *melcor__pk* datatype was converted from.

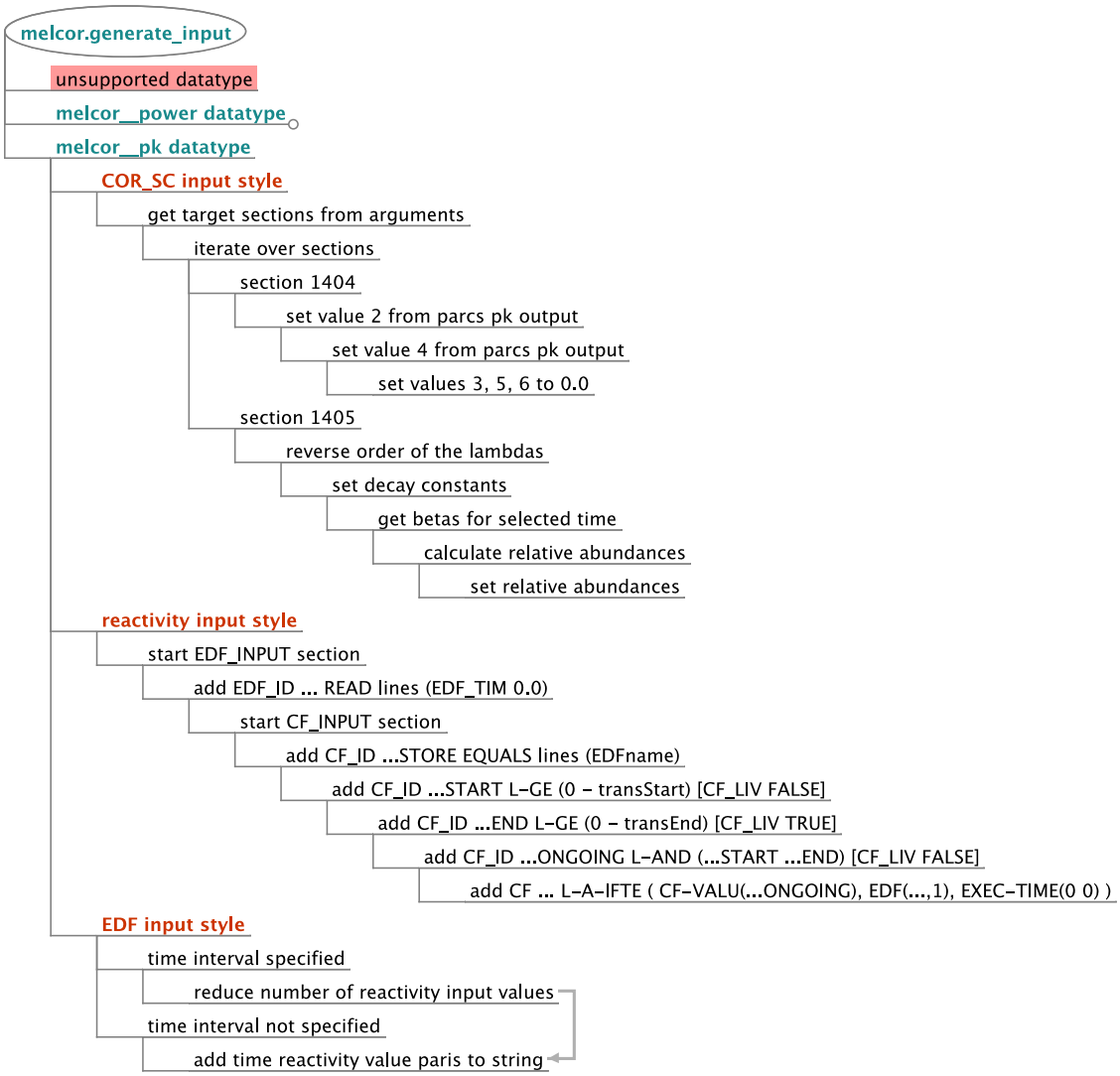

Figure 5.6: The program flow of the melcor.generate_input routine. This algorithm focuses on the generation of point kinetics parameter input generation.

The 'reactivity' input style generates a number of MELCOR EDF and CF inputs which can be used to easily connect a CF value to an EDF. A total of one EDF function and five control functions are created by this input generation algorithm. The generated control functions are used to: determine when to switch on the effect of th EDF, determine when to switch it the EDF values, and provide a single CF through which all of the logic is abstracted.

The 'EDF' input style generates a MELCOR EDF containing transient reactivity values as a function of time. These values may be easily connected to the MELCOR deck use of the aforementioned 'reactivity' input generation style. Additionally, since MELCOR automatically interpolates between points of an EDF, it may be desirable to reduce the number of independent-dependent variable pairs in the output. If specified with a time interval, this point reduction will also occur.

5.1.7 The melcor.load_input Function

The algorithm depicted in figure 5.7 is a representation of the operational progression of the function responsible for loading a MELCOR input file. This diagram depicts cases of multiple possibilities as vertical lines joined by two or more connecting horizontal lines. The entry point for this diagram is from the top-left bubble.

From this diagram it can be seen that, this operation proceeds for the most part in a linear fashion. Essentially, if this function is not supplied with an invalid path, it will load all of the data from the input file and return *melcor__input* datatype. This datatype is useful in that it can be used in conjunction with the melcor.update function.

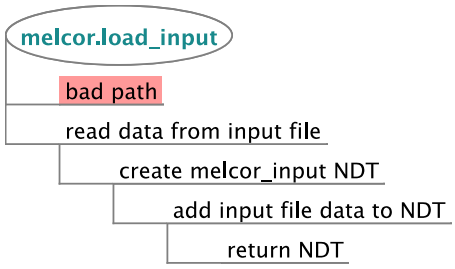

Figure 5.7: The program flow of the melcor.load_input routine.

After any number of modifications have been made to the input file by used of the melcor.update function, the resultant input file can be created by using the melcor.view function on the datatype to produce a string and then subsequently,

using the fs_write function to write the string to a file.

5.1.8 The melcor.run Function

The algorithm depicted in figure 5.8 is a representation of the operational progression of the function responsible for starting up MELCOR. This diagram depicts cases of multiple possibilities as vertical lines joined by two or more connecting horizontal lines. The entry point for this diagram is from the top-left bubble.

From this diagram it can be seen that, there are three possible failure modes for this function. All of these particular failure paths will result in the production of their own unique error message. Any error message produced by a function of an external module is captured and reported by the ovr interpreter after the completion of a toplevel *evaluate line* sequence.

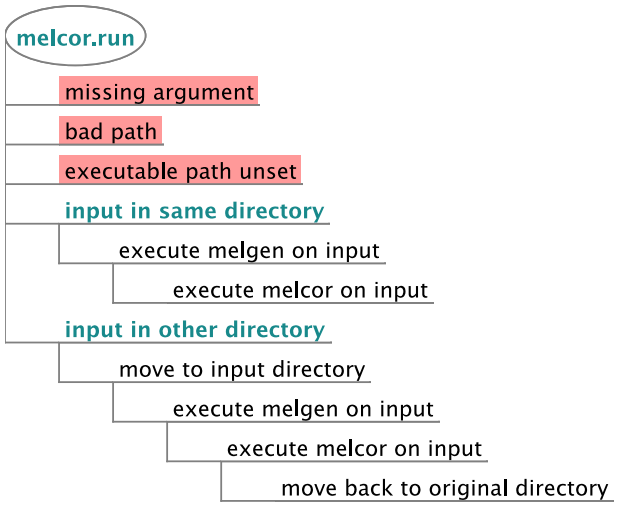

Figure 5.8: The program flow of the melcor.run routine. Nodes with red backgrounds represent instances of failure and result in termination of the code execution (if in script mode).

It can be seen from the preceding diagram, that the execution of the binary always takes place within the folder that contains the target input file, this is the desired behavior for a couple of reasons. At least one of the codes being coupled

to in this research had an insufficient character limit for its target input argument. This character limit was such that, calling the executable with an absolute path could cause the program to fail (depending on where an input file was located). The second reason for this behavior is so that the output files for a particular program execution end up being created in the same folder as their corresponding input file.

The last interesting aspect to point out this algorithm is that this one command executes both MELGEN and MELCOR and in that order. This is done because the MELGEN executable essentially creates the initial restart file from which MELCOR execution can begin.

6. OVERSEER IMPLEMENTATION

Overseer is a coupling code designed for use in the nuclear engineering field. The interface of this code is modeled after the Python programming language. This program provides a set of tools to simplify the translation and passing of information between its supported computational codes. Additionally, overseer provides functions to automate the launching of codes and the plotting of their output. The name of this program implies its role and purpose. Generally speaking, an overseer is someone who directs the work of others. In the case of this program, it is computational codes that are directed.

6.1 Design Features

6.1.1 Input Format

6.1.1.1 In code comments

Line comment:
To comment out an entire line of code, prepend a '#' at the start of a line. In the case of a line comment, preceding indentation does not matter. The following are two valid examples:

```
# this is a comment line
    # another comment line (indented)
```

String comment:
Another style of in-code comment is a string style comment. To use this style, simply include a line which contains a single valid OVR string. OVR will evaluate the string to make sure that it is valid, but beyond this, it will do nothing. String comments using either type of quotation mark are valid, for example:

```
"a simple string comment"
```

```
'second style of string comment (using single quote mark)'
```

Just as with line-comments, string comments can be indented an arbitrary amount, without causing problems.

6.1.1.2 Location Strings

Current working directory: The simplest location possible is one within the current working directory (CWD). The CWD is normally where execution of OVR or an OVR script commenced.

```
# --- EXAMPLE: writing a file to the CWD
>> fs_write("hello.txt", "greetings...")
```

The fs_write function is used to write files to a location. In this example only a filename was supplied, so the resulting file will be in the current working directory. Overall, this command creates a file called "hello.txt" which contains the text "greetings..."

Absolute paths: When a path includes information starting from the root directory (potentially of a device) to that of the target location, it is an absolute path. These absolute paths tend to be longer, but are generally more predictible than using current working directories. Also, in some situations it is easier to specify the absolute path of a location as opposed to using a realative location. On all platforms the path separator that should be used to describe such a location is, '/'.

```
# --- EXAMPLE: writing to an absolute path (on Windows)
>> fs_write("c:/somedir/atest.txt", some_str)
# --- EXAMPLE: writing to an absolute path (on Linux/OS X)
>> fs_write("/somedir/atest.txt", some_str)
```

Relative paths: These paths are specified relative to the current working directory (CWD). The benefits of relative paths are that: they tend to be shorter than equivalent absolute paths; and if all inputs are located inside of a directory or subdiretories thereof that directory may be moved without affecting the relative paths.

```
# --- EXAMPLE: Single nested directory
>> parcs.run("SS-example/parcsinput")
```

Here, the function parcs.run is used, it runs PARCS on the supplied input file. In this case the input file ‘parcsinpup‘ is located in a directory ‘SS-example‘, which was inside of the CWD.

```
# --- EXAMPLE: Moving up a directory
>> parcs.set_path("../parcs/BIN/agree.exe")
```

In this example, parcs.set_path was used, this function sets the expected location of the PARCS executable. In this case, the ‘parcs‘ directory was located one level up from the CWD. To move upwards in location heiarchy, a ‘..‘ must be included in the path as if it were a folder. A relative path may move up multiple directories by using multiple entries of ‘..‘ as directories at the start, as long as the resulting path is valid.

6.1.1.3 Multi-line lists

To improve readability of overseer code input, lists may be broken up over multiple lines. For this to work, a line of a multi-line list must end with a comma (excluding the last line). A couple examples of this formatting convention are as follows:

Simple lists: These lists are used by assigning them to a variable.

```
somelist = ["object1",
            "object2", "object3",
            778.6]
# --- which is equivalent to the following
somelist = ["object1", "object2", "object3", 778.6]
```

An argument list A second situation where multi-line lists may be used is for function arguments. Essentially, a function is called with a list of arguments, this treatment allows for funtion arguments to be broken up over multiple lines.

```
print("The quick brown fox jumps over the lazy dog.",
      "object2", "object3",
      778.6)
```

The print function can be called with any number of arguments, these arguments are printed out one by one with a space inbetween them. So the result of this input is the following

```
The quick brown fox jumps over the lazy dog. object2 object3 778.6
```

6.1.2 Error Reporting

One of the design goals of overseer was to fail gracefully and to produce detailed error messages in the case of an input error. The motivation behind this goal is to help users to find and fix input mistakes faster and also to reduce the learning curve for the program.

To achieve this goal, Overseer includes a number of internal routines that check for errors at low, medium and high program operational levels and report the cause of the error, each from their respective levels. By providing multiple error messages generated at various points of failure for the interpreter the user gets a more detailed idea of what caused the error. Below is a sample of an error message produced by a single line of bad input.

```
# --- EXAMPLE: error during interactive session
>> cat - 1
ERR: calculation failed ‘cat - 1‘
ERR: mathematical calculation impossible between ‘None‘ and ‘1‘
ERR: Undeclared variable ‘cat‘
```

Initial errors can only be triggered at the lowest level of program operation. However, if a lower level routine places an error on the stack before it completes, it will cause subsequent medium and high level routines to fail and report an error

to the stack. The highest level of the operational routine is what actually handles the error stack. If Overseer was being run as an interactive session, all errors of this nature are recoverable and the user can correct their mistake and continue. Consequently, if Overseer is interpreting from a file, errors of this nature will cause a termination of the code and an additional piece of information regarding the location of the error.

```
# --- EXAMPLE: error while interpreting a script
ERR: Function call failed ‘parcs.set_path("/some/wrong/location/parcs.exe")‘
ERR: Nonexistant file or bad path ‘/some/wrong/location/parcs.exe‘
    on line 4: ‘parcs.set_path("/some/wrong/location/parcs.exe")‘
```

When the highest level routine is handling a populated error stack, the order of the messages are reversed before reported. In this manner the user starts out with the most macroscopic view of the error, and with each subsequent message the report gets more specific about the nature of the error.

6.1.3 Modular Design

The structure of the coupling code is in the form of a general purpose main module, some general support modules, and other code specific features added by additional optional modules. The main reasons behind this code structure were: to reduce the possibility of introducing errors during development as well as, to allow for individual code modules to be easily disabled.

While direct interactions are allowed between the optional modules and the main or support modules, they are not allowed directly between two optional modules. Information passing between the optional modules can occur only by passing output run time variables from one optional as input to another optional module's function. By having each optional module adhere to their own particular static API, there is improved ability for the development of individual modules without unexpectedly

adversely affecting the operation of others.

Another capability of the code as designed is for the easily disabling and removing the various individual optional modules. The reason for adding this particular capability is to allow for the easy removal of interfacial capabilities to potential export controlled codes, should they be included at some point in the future. This would allow the resulting compiled code to be distributed to a larger academic audience without disseminating capabilities which can't be easily shared.

6.1.4 Output Post-processing

Another important design feature for this code was the inclusion of a number of output post processing capabilities. The most developed of these features by far, is the interface for data plotting using gnuplot. The other output processing features include a viewer and a general melcor.get function.

All of these capabilities are demonstrated in a general sense in the following example:

```
# ----- MINIMUM VERS. ovr 2.70
#       parcs/melcor view/graph example

#--- set path to gnuplot binary (required for plotting)
report.set_gnuplot_path("C:/mleimon/gnuplot/binary/gnuplot.exe")

# --- load output corresponding to preceding melcor input file
output_mel = melcor.load_output("input--melcor/pbmr400.inp")

# === usage of melcor.get()
# --- extract and print the simulation times available in the melcor output file
times = melcor.get(output_mel, "avg_comp_temps", "keys")
print(times)
```

```
# --- extract and view a spatial temperature distribution
ATemp = melcor.get(output_mel, "avg_comp_temps", 9002.0)
print()
print(melcor.view(ATemp))

melcor.graph(ATemp, ["rings", 2, 6], ["levels", 6, 26], ["style", "map"],
["name", "test-map"])

melcor.graph(ATemp, ["rings", 2, 6], ["levels", 6, 26], ["style", "3d"],
["name", "test-3d"])
```

The code above is the input to the example. The corresponding output for this example is as follows:

```
loading output generated from ‘input--melcor\pbmr400.inp‘ file
adding ‘input--melcor\pbmr400.out‘ to data
[0.0, 25.003, 50.0, 75.022, 100.01, 200.0, 300.01, 400.01, 500.02, 600.01,
700.0, 800.02, 900.01, 1000.0, 1100.0, 1200.0, 1300.0, 1400.0, 1500.0, 1600.0,
1700.0, 1800.0, 1900.0, 2000.0, 2100.0, 2200.0, 2300.0, 2400.0, 2500.0, 2600.0,
2700.0, 2800.0, 2900.0, 3000.0, 3100.0, 3200.0, 3300.0, 3400.0, 3500.0, 3600.0,
3700.0, 3800.0, 3900.0, 4000.0, 4100.0, 4200.0, 4300.0, 4400.0, 4500.0, 4600.0,
4700.0, 4800.0, 4900.0, 5000.0, 5100.0, 5200.0, 5300.0, 5400.0, 5500.0, 5600.0,
5700.0, 5800.0, 5900.0, 6000.0, 6100.0, 6200.0, 6300.0, 6400.0, 6500.0, 6600.0,
6700.0, 6800.0, 6900.0, 7000.0, 7100.0, 7200.0, 7300.0, 7400.0, 7500.0, 7600.0,
7700.0, 7800.0, 7900.0, 8000.0, 8100.0, 8200.0, 8300.0, 8400.0, 8500.0, 8600.0,
8700.0, 8800.0, 8900.0, 9000.0, 9000.1, 9000.2, 9000.3, 9000.4, 9000.5, 9000.6,
9000.7, 9000.8, 9000.9, 9001.0, 9001.1, 9001.2, 9001.3, 9001.4, 9001.5, 9001.6,
9001.7, 9001.8, 9001.9, 9002.0, 9002.1, 9002.2, 9002.3, 9002.4, 9002.5, 9002.6,
9002.7, 9002.8, 9002.9, 9003.0, 9003.1, 9003.2, 9003.3, 9003.4, 9003.5, 9003.6,
9003.7, 9003.8, 9003.9, 9004.0, 9004.1, 9004.2, 9004.3, 9004.4, 9004.5, 9004.6,
9004.7, 9004.8, 9004.9, 9005.0, 9005.1, 9005.2, 9005.3, 9005.4, 9005.5, 9005.6,
9005.7, 9005.8, 9005.9, 9006.0, 9006.1, 9006.2, 9006.3, 9006.4, 9006.5, 9006.6,
```

9006.7, 9006.8, 9006.9, 9007.0, 9007.1, 9007.2, 9007.3, 9007.4, 9007.5, 9007.6,
9007.7, 9007.8, 9007.9, 9008.0, 9008.1, 9008.2, 9008.3, 9008.4, 9008.5, 9008.6,
9008.7, 9008.8, 9008.9, 9009.0, 9009.1, 9009.2, 9009.3, 9009.4, 9009.5, 9009.6,
9009.7, 9009.8, 9009.9, 9010.0, 9060.0, 9110.0, 9160.0, 9210.0, 9260.0, 9310.0,
9360.0, 9410.0, 9460.0, 9510.0, 9560.0, 9610.0, 9660.0, 9710.0, 9760.0, 9810.0,
9860.0, 9910.0, 9960.0, 10000.0]

MELCOR average component temperature (time =9.0020e+03 s):
x-axis (radial rings): 1-8 [left->right]); y-axis (axial levels): 1-29 [top->bottom]

893.15 893.61 893.57 893.51 893.22 892.18 884.81 825.67
789.35 788.04 785.35 785.35 785.34 785.31 781.26 775.38
901.51 918.44 935.55 935.56 935.56 935.58 773.66 772.91
873.44 1079.23 1111.6 1111.61 1111.62 1111.8 774.19 772.75
848.64 1167.58 1167.67 1167.75 1167.88 1167.02 798.78 772.77
872.57 1101.63 1116.74 1142.14 1183.86 1305.07 940.14 823.52
874.87 1101.03 1115.91 1141.55 1183.99 1313.71 954.76 828.89
874.08 1100.69 1115.13 1140.67 1182.88 1314.09 955.13 829.14
872.62 1099.86 1113.79 1139.18 1181.07 1313.52 953.36 828.64
870.74 1098.5 1111.87 1137.04 1178.52 1312.01 950.74 827.87
868.41 1096.56 1109.29 1134.19 1175.16 1309.44 947.28 826.84
865.55 1093.96 1105.95 1130.52 1170.85 1305.67 942.87 825.52
862.1 1090.55 1101.72 1125.87 1165.41 1300.45 937.35 823.85
857.94 1086.14 1096.37 1120.0 1158.6 1293.44 930.56 821.78
853.0 1080.46 1089.63 1112.62 1150.06 1284.19 922.31 819.27
847.18 1073.11 1081.1 1103.3 1139.36 1271.99 912.44 816.25
840.46 1063.61 1070.29 1091.54 1125.91 1256.18 900.83 812.68
832.82 1051.31 1056.6 1076.66 1109.0 1235.43 887.43 808.56
824.36 1035.5 1039.33 1057.97 1087.88 1208.71 872.37 803.91
815.28 1015.38 1017.77 1034.69 1061.74 1174.79 855.97 798.84
805.92 990.2 991.25 1006.15 1029.86 1132.48 838.81 793.52

```
796.81 959.38 959.32 971.89 991.8 1081.0 821.81 788.24
788.63 922.85 922.05 932.0 947.69 1020.34 806.13 783.35
782.11 881.4 880.36 887.48 898.65 951.82 793.1 779.28
777.67 840.62 839.76 844.18 851.08 884.56 783.74 776.35
774.86 812.11 811.45 813.98 817.94 837.62 777.93 774.51
773.27 793.38 792.88 794.19 796.23 806.81 774.58 773.41
773.03 0.0 0.0 0.0 0.0 0.0 773.14 772.19
773.0 773.28 773.27 773.29 773.32 773.5 773.0 772.94
image ‘test-map.png‘ created
image ‘test-3d.png‘ created
```

So it can be seen that the melcor.get function could be very well used to determine what sort of simulation times are available for extraction. Furthermore, the individual spatial temperature information for a given time can either be plotted with the graph function or viewed with the view function. The view function is very useful in the event the user wants to extract output data and then import it into a spreadsheet program. It is even possible to swap the axis of the tabular output with the view function (e.g. setting the x-axis to axial levels)

In general the graph functions allow for some amount of customization of the output. In this example, the ‘rings’, ‘levels’, ‘style’ and ‘name’ options of the plot were all manually specified. Notice that the resulting plots only include a subsection of the total spatial temperature nodalization, this was due to the specification by the ‘rings’ and ‘levels’ options. This selection was made because this smaller region includes only the core region and not the supporting structures. Both of the resultant plots are included below so that their plot styles can be seen and compared.

The following figure (figure 6.1) is one of the two plots generated by the input at the start of this section. This particular plot was generated by the melcor.graph function that used the "map" style optional argument. This plot contains three dimensions of information, namely: average component temperatures, axial levels, and radial rings. The value of the temperature for a particular cell is depicted by the color of the cell. A cell temperature value can be determined approximately by locating it's color within the key on the right.

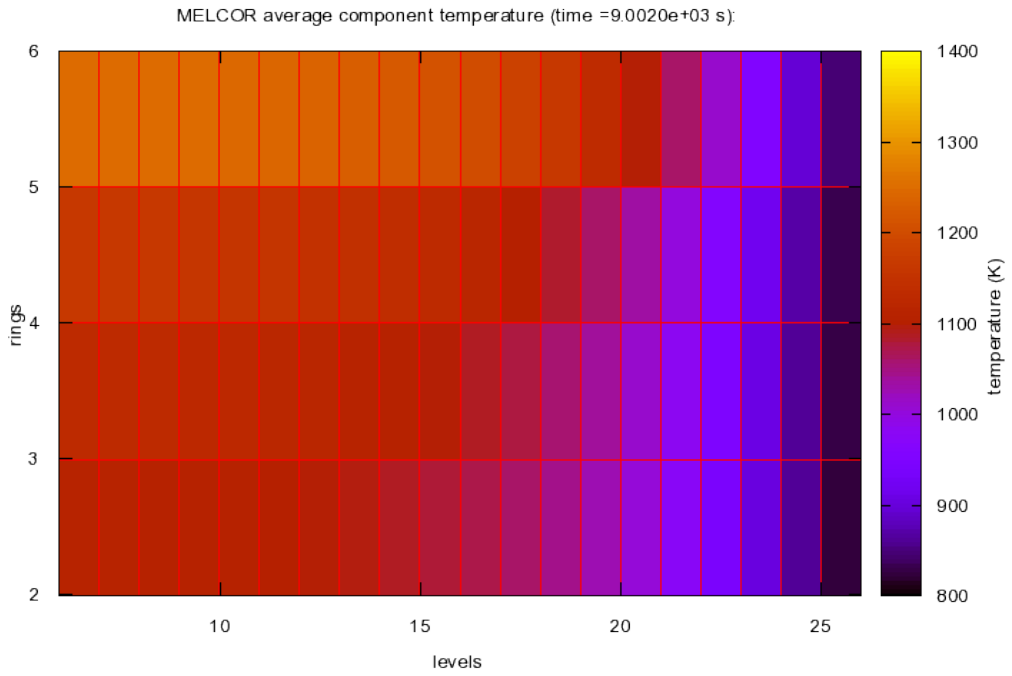

Figure 6.1: This is the plot produced of ATemp using the value "map", for the 'style' option.

The information plotted by this function is the average spatial component temperatures, as extracted from the "EDIT OF CORE CELL AVERAGE TEMPERATURES (K)" section of the corresponding MELCOR output file.

The following figure (figure 6.2) is one of the two plots generated by the input at the start of this section. This particular plot was generated by the melcor.graph function that used the "3d" style optional argument. This plot contains three dimensions of information, namely: average component temperatures, axial levels, and radial rings. The value of the temperature for a particular cell is depicted in this plot in two different ways: by the color of the cell as well as the location of the cell in the z-axis of the plot. A cell temperature value can be determined approximately by locating it's color within the key on the right.

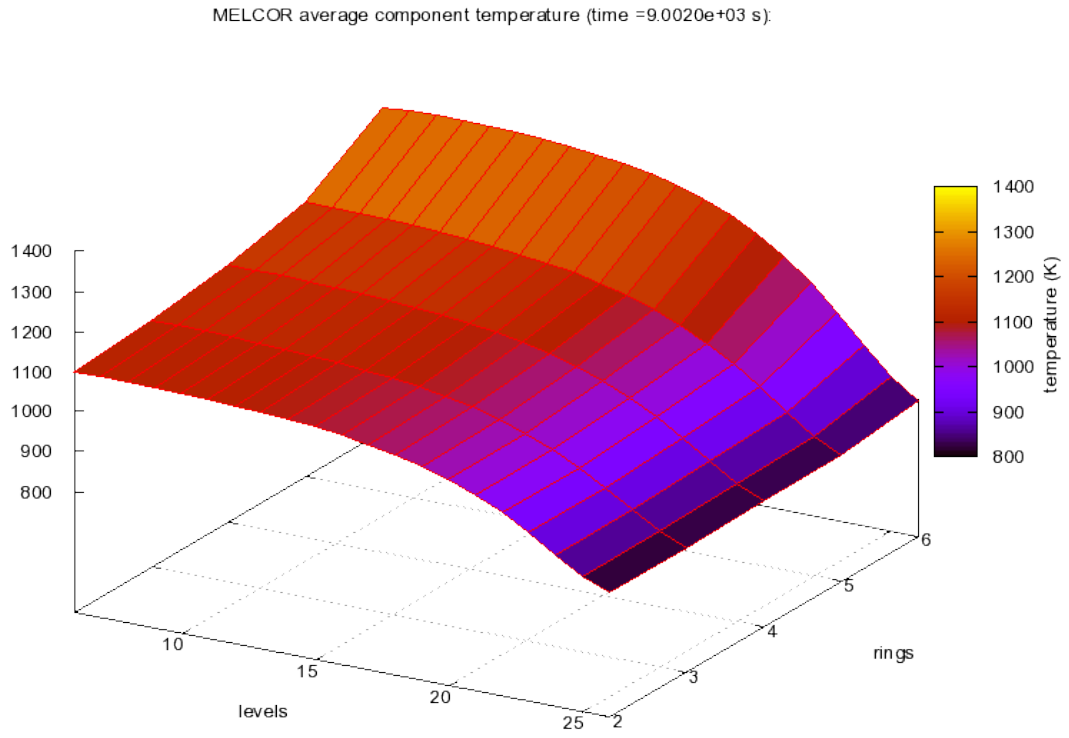

Figure 6.2: This is the plot produced of ATemp using the value "3d", for the 'style' option. This is also the default output style for any 3d plot. Therefore omission of the 'style' option will produce a plot like this.

As with the previous plot, the information plotted by this function is the average

spatial component temperatures, as extracted from the "EDIT OF CORE CELL AVERAGE TEMPERATURES (K)" section of the corresponding MELCOR output file.

Additionally, to speed up the analysis process for large output files, the graph function can work directly with the output native datatypes. This allows the user to specify which types of plots they would like, and then with a single command, plot all of them for all available output simulation times. Further demonstration of these capabilities are outlined in the demonstrations chapter of this thesis.

6.2 Information Mapping

6.2.1 Core to Core Power Mapping

In general, special care must be taken when designing the input files for a pair of simulation codes if the desired goal is some sort of coupling between them. While there are expected to be a number of differences between the models due to differences in the capabilities of each code, it is important that the details most closely linked to the desired coupled parameter be equivalent. For the test case of this coupling code, one such equivalent detail is the nodalization of the core region, as seen in figure 6.3. With this particular issue addressed, it is possible to address another detail of the coupling, the mapping of these equivalent regions.

The image depicted in figure 6.3 is a scaled representation of the spatial nodalization used in the PARCS/AGREE model for the PBMR400. This visual is a two-dimensional radial slice of the model. The three dimensional volume is created by the rotation of the two-dimensional slice around radial center point of the model (the left side of the model shown here). This is essentially how the PARCS/AGREE model specifies the nodalization for the core region, a single pair of axial and radial grids are specified and the pair is applied to a 360° rotation about the z-axis.

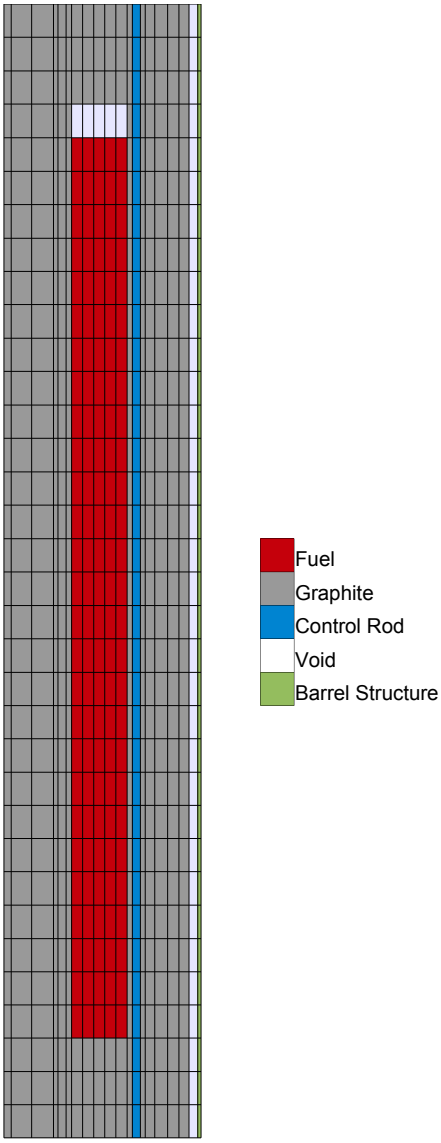

Figure 6.3: A correctly scaled view of the PARCS/AGREE nodalization for the PBMR400 input model. All coupled spatial power output comes from the regions shown in red here. Additionally, these fuel regions have an equivalent nodalization in the MELCOR model.

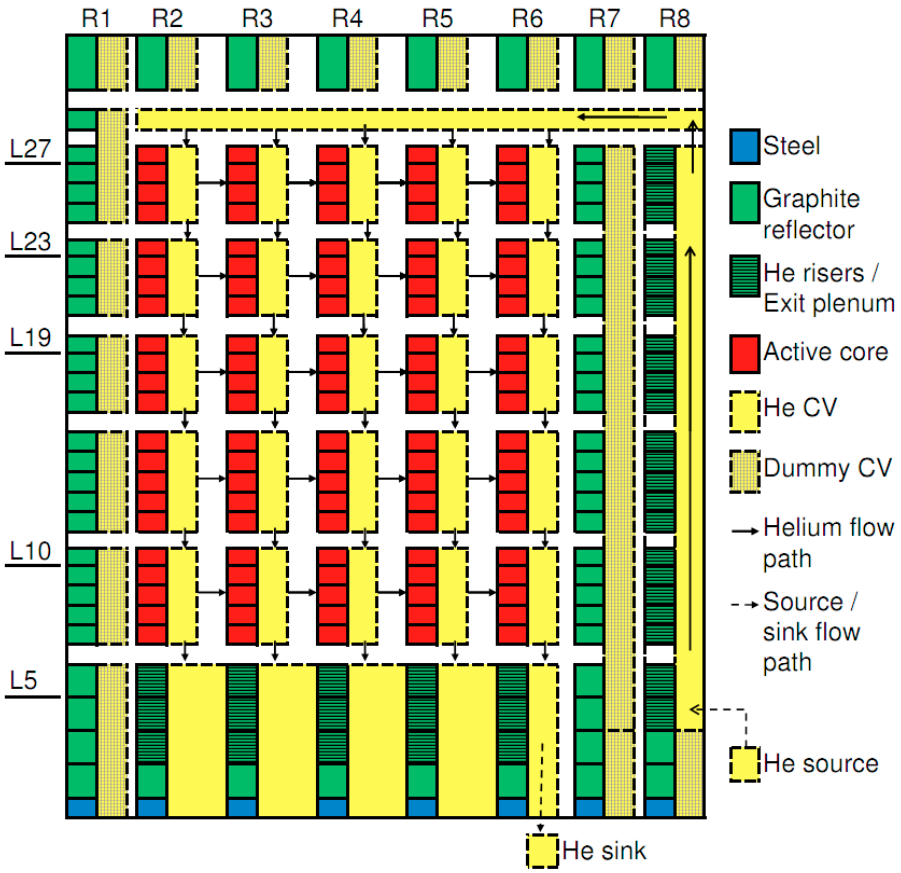

Figure 6.4: A color coded COR diagram for the MELCOR model of the PMBR400 used in the coupled simulations. The core fuel regions of this model are also shown here in red (axial levels 6 - 27 for radial rings 2 - 6). (Bradley Beeny, personal communication, October 5, 2012)

In keeping with the fundamental design ideals of simplicity and extensibility, a simple and somewhat generic spatial power coupling routine was devised to accomplish the mapping of these equivalent regions. This coupling routine utilizes a list of self contained powermaps, which will henceforth be referred to in this document as maplists. For the PARCS/AGREE to MELCOR case, each individual powermap describes a block of spatial locations (exactly as they would appear in a 'parcs.map' output file) as well as a corresponding starting MELCOR axial level, radial ring pair.

In more detail, the block of spatial locations in PARCS/AGREE is described

adequately by specifying an extraction time, a start location, and an end location. Each of these locations are lists or tuples of two integers which represent an (x, y) index from a particular output power table for the specified time. The upper-left most value entry in a table has an index of (0, 0), and the x and y indices, increase as we move right from column to column and down from row to row, respectively. If a starting index for a particular axis is higher than a corresponding ending index, then the data will be reversed in that axis when it is translated into a corresponding MELCOR spatial power datatype. Fig 6.5 depicts a block of spatial locations in a PARCS/AGREE output as defined by two locations.

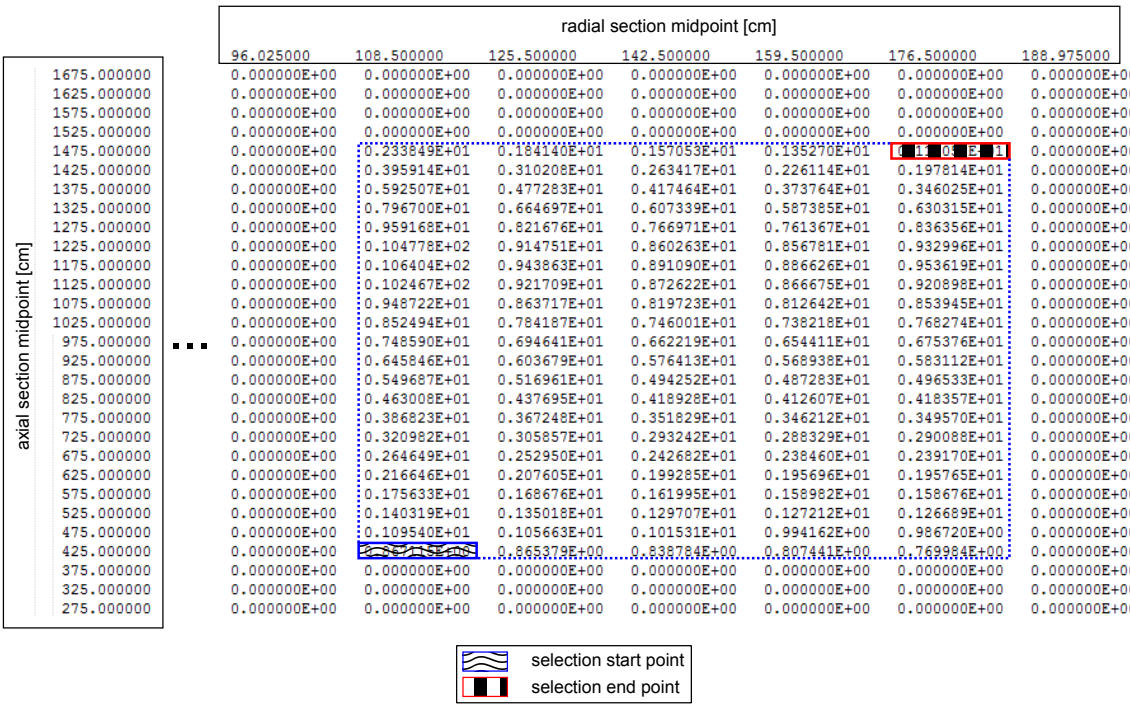

Figure 6.5: A illustrated power selection from a PARCS output file. The power selection is described as nodes bounded by two specified locations, a start point and an end point. Here it can be seen that the starting index value in the y-axis is higher than the y-axis location of the ending point (25 versus 4) and so, the axial levels data will be reversed when it is translated into a corresponding MELCOR spatial power datatype.

The MELCOR side of the mapping is more simply defined by a starting radial

ring, axial level, and a starting control function value. For spatial power coupling from PARCS/AGREE to MELCOR, values for control volume power are calculated from corresponding PARCS/AGREE power densities and their corresponding volumes, which are calculated from the geometrical description of the nodalization in the PARCS/AGREE input file. The index location y-axis of the PARCS/AGREE output corresponds to MELCOR axial levels, except that these have opposite directions. The index location x-axis of the PARCS/AGREE output corresponds to MELCOR radial rings. Since a number of MELCOR control functions may need to be created to specify the power to MELCOR, the user gets the option to specify a control function number, from which enumeration of these new control functions will occur. This is done so that the user can avoid redefining a control volume previously defined.

```
powermap = [(0.0,(6,25),(10,4)), (2, 6, 400)]
#          [(parcs description), (mel desc )]
# parcs description : (extract_time, (loc_start), (loc_end))
# melcor description: (ring_start, level_start, CF_start)
```

Figure 6.6: A sample powermap with a visual breakdown of its contents. It extracts power density values from a PARCS/AGREE simulation time of 0.0 s. The block being extracted is the same as the one depicted in Fig 6.5. When these values are used to generate MELCOR power input, the axial levels from PARCS/AGREE will be reversed. More specifically, axial level 25 from PARCS/AGREE becomes axial level 6 in MELCOR, axial level 24 from PARCS/AGREE becomes axial level 7 in MELCOR, et cetera.

The extensibility aspect of this mapping implementation stems from the ability to specify multiple powermaps for a single coupling. This should allow for relatively straightforward coupling of cores with a more complex geometry than the one used in the demonstrations here. If the user does decide to use multiple powermaps, it should be noted that only the first control function start value is used and so, all subsequent

maps will simply iterate upwards in control function numbering from where the previous powermap stopped. This removes the burden of correctly calculating the number of control functions for a more complex core coupling.

6.2.2 Transient Event Mapping

Since computational codes may operate in fundamentally different ways, it may be necessary for a code or codes to run initially for a time to reach approximately steady state. To accommodate for this potential difference between the codes, when a point kinetics information conversion function is called, it must be called with equivalent simulation times for both codes. As a part of the function's operation, it shifts the time of the reactivity values by the appropriate amount to match the desired time for the output target.

Another optional adjustment which can be made by the user is a reduction of the number of output transient data points. If for instance, the input code happens to operate with a very small time step and the output code does not, it might be beneficial to reduce the number of output transient points. While the converted point kinetics information will contain all of the data points, a time interval may be optionally specified when calling the MELCOR point kinetics input generation routine. In this case, there will be a minimum time interval between output data points, resulting in a net reduction of output transient data points.

6.3 Overseer Layout

As was mentioned previously, the coupling code was designed with modular construction in mind. In general, each module or part of the code, provides a number of specific datatypes and or functions to help accomplish various coupling related tasks. This section outlines the structure and functionality of the most fundamental part of the coupling code, the basic overseer interface.

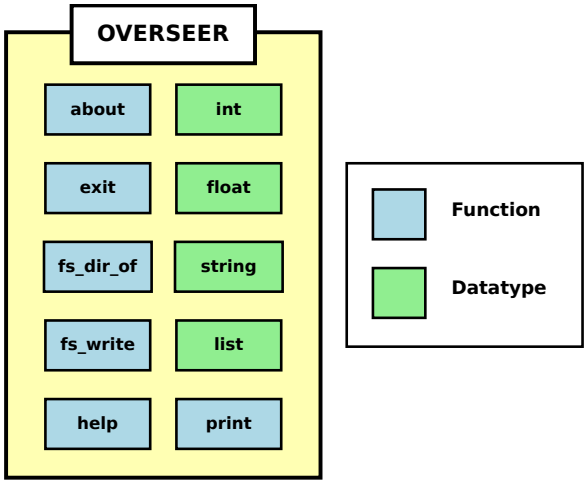

Figure 6.7: A graphical representation of the structure of the basic Overseer interface.

6.3.1 Basic Datatypes

Without using the aid of any external functions, an ovr user is capable only of creating and manipulating a number of fundamental datatypes. These fundamental datatypes include: floats, integers, lists and strings. These fundamental datatypes are used by all of the other modules of the code.

These fundamental datatypes are used by many different functions that find standard usage in any coupling accomplished by ovr, more specifically though not exclusively: float values are used to specify time intervals and time steps; integer values are used to specify power conversions inside of powermaps and also find usage in some graph options; the list datatype is used to convey sets of data such as powermaps or ZP,RP tables; strings are used by every load_input and load_output function of the supplied modules as well as in many other places.

The rest of this section is dedicated to a more detailed explanation of these basic ovr datatypes.

6.3.1.1 Float

From a practical standpoint, Floats may be treated as numbers which require a decimal point to describe correctly. In actuality, this particular datatype is technically much more complex. Deriving from ovr's Python 3 lineage, it is also subject to the same sort of limitations with this datatype. One important aspect to note is that, floating points numbers are not exact representations of decimal numbers. Generally for a 64bit floating point number, approximately only the first 16 digits of the decimal number are likely correct.

In ovr, floats can be either negative or positive. To create a variable containing a float as opposed to an integer, all that is required is that a period and a trailing zero are appended to the value.

One place where an ovr user will be required to use a float to accomplish a coupling is in the process of spatial power conversion from PARCS/AGREE to MELCOR. When spatial power information is loaded from PARCS/AGREE output, all of the information extracted is stored internally in blocks which encapsulate the individual timesteps. To properly select a timestep from which a spatial power profile should be extracted, a valid and present float value must be specified within the powermap being used for the conversion. Another place float values are used is within the tab_ZP,RP lists. It may be necessary to supply the elevations, porosities and radii with float values.

Example: Defining a float

```
>> x = 5.0
>> print(x)
5.0
```

Another thing to note is that, the resulting value from a float being multiplied with or added to an integer value, is always another float.

Example: Float-integer operations

```
>> print(x*2)
10.0
>> x = x + 3
>> print(x)
8.0
```

6.3.1.2 Integer

Nondecimal or whole numbers are integers. In OVR, if a supplied number is valid and does not include a deimal with trailing values, it will be stored as an integer. Integers may be either positive or negative. As is the case with Python 3, OVR does not have a size limit for integers. If an operation would cause a value to exceed its current memory allocation, Python 3 automatically reallocates more memory to store the value.

The user should also anticipate the use of integer values when accomplishing spatial power conversion from PARCS/AGREE to MELCOR. Within powermaps, integers are used to specify PARCS/AGREE output data indices as well as the starting radial ring, axial level and CF ID. Additionally, this datatype finds usage with the melcor.generate_input function, the COR_ZP,RP list descriptions, and in optional arguments for the melcor.graph function.

Example: Integer math

```
>> x = 10
>> y = -x + 3
>> print(x, y)
10 -7
```

6.3.1.3 List

Another basic datatype provided by OVR is the list. In OVR, lists are a datatype which may contain any combination of integers, strings, floats and even other lists.

From the user's standpoint, lists may be created by placing a number of comma separated values within a pair of corresponding square brackets. When processing a list, OVR evaluates all elements of the list sequentially and individually. The major benefit of this being that list elements may be or include variable references and may additionally include any valid mathematical operations that OVR is capable of processing.

In the process putting together a coupling procedure, the user will be required to utilize lists in a number of places. Generally, lists are only required when a large or variable quantity of data must be encapsulated within a single object. This occurs both with specifying COR_ZP,RP tables as well as with powermaps.

A number of examples of list creation and usage are presented below.

Example: Defining an empty list

```
>> l = []
>> print(l)
[]
```

In this example an empty list is created. This type of list has limited use, but is certainly possible.

Example: List containing various types:

```
>> l = ["a", 123, 77.4]
>> print(l)
['a', 123, 77.4]
```

One of the large strengths of lists is that they may contain various datatypes. This allows the user to lump together a number of related values even if they are not of the same type.

Example: A list containing other lists:

```
>> l = ["a", 123, 77.4]
```

```
>> lil = [l, l]
>> print(lil)
[['a', 123, 77.4], ['a', 123, 77.4]]
```

Here, a previous list 'l' was referenced and from it a new list was created with two instances of 'l' as elements.

Example: List addition:

```
>> x = [1, 2, 3]
>> y = [4, 5, 6]
>> z = x + y
>> print(z)
[1, 2, 3, 4, 5, 6]
```

When adding lists, the result is that the elements of succeeding lists are appended to the end of the preceding lists.

6.3.1.4 String

In OVR, the string datatype is used to contain a variable number of ASCII characters. Strings may be created by surrounding text information by either single or double quotation marks. For the string to be valid, both starting and ending quotation marks must be of the same type.

This datatype is likely the most used of all the fundamental ovr datatypes. Strings are used to specify locations for all of the load_input/output functions as well as with all of the run procedures. Additionally, strings are also often used to specify optional arguments or operational styles if they are supported by a particular function.

Example: Some basic strings

```
>> style1 = "a valid string"
>> style2 = 'also valid'
>> print(style1, style2)
a valid string also valid
```

Example: Addition of strings

```
>> prefix   = "VAR_PREFIX_"
>> obj_name = prefix + "18"
>> print(obj_name)
VAR_PREFIX_18
```

Addition of strings can be quite useful if a material or variable in another code has a long prefix name. In which case, the user can supply variables by combining the prefix and enumeration.

Example: Strings containing quotation marks

```
>> complex = "The house's " + '"cat" meowed.'
>> print(complex)
The house's "cat" meowed.
```

Another use of string addition is to create strings which contain both types of quotation marks. This technique may necessary in a situation where a computational code's input style uses either of these quotation characters.

6.3.2 Functionality

A number of general and basic functions are provided OVR. Since these general functions are in no way specific to any particular computational code, they exist inside of the main coupling code itself. From the user's standpoint, this means only that a module name is not required when calling one of the basic OVR functions. The rest of this section is dedicated to a more detailed explanation of these basic OVR functions.

6.3.2.1 exit()

```
exit( optional:{int return_val} )
```

Returns: *nothing*

This function terminates the overseer shell. If called with an integer argument, it will return this code at exit. Otherwise by default, it will return a value of zero (no error).

Example: Exiting with an error

```
>> exit(2)
```

This command causes the overseer shell to exit with an exit code of '2'. Remember that it is good practice to only return non-zero value when the a program has exited abnormally.

6.3.2.2 fs_dir_of()

```
fs_dir_of( string location )
```

Returns: *string*

Given a string containing an absolute or relative path file location, this function will return a string representation of the absolute path that contains the file. This function can be used to simplify the process of accessing or creating files in the same directory as another file that had it's location specified previously.

Example:

```
>> some_file = "/some/deeply/nested/input.txt"
>> path = fs_dir_of(some_file)
>> other_file = path + "other.txt"
>> print(other_file)
/some/deeply/nested/other.txt
```

Logically, the next step following this example might be to write to or load something from this other file location however, that is not detailed here. Note that, OVR will check to make sure that the file specified actually does exist, if the file path is invalid, an error will be reported.

6.3.2.3 fs_write()

```
fs_write( string file_path, string data optional:{overwrite} )
```

Returns: *nothing*

This function writes the contents of the string 'data' to a file location indicated by the string 'file_path'. The 'file_path' string may include path information. This path information can either be absolute or relative in nature. Omission of path information will result in a file location inside of the current working directory. The optional last argument 'overwrite', serves as an automatic overwrite toggle. The default behavior of OVR is to prompt the user when attempting to overwriting an existing file. To overwrite files automatically, set the optional argument to '1' or 'overwrite'.

Example: Writing to the current working directory

```
>> fs_write("test.txt", "this is the content of this test text file")
```

To write a file to the current working directory (the directory from which OVR execution was started) just set the location argument to the desired filename. In this example, we have created a file named 'test.txt' and it contains the following single line of text, 'this is the content of this test text file'.

Example: Writing to an absolute path

```
>> fs_write("c:/somedir/atest.txt", some_str)
```

To write to an absolute path, the string which contains the filename must be preceded by the directories in which it should reside. Notice that the path separator used in this windows path is a forward slash, '/'. While this is not the path separator ordinarily used with windows, this convention allows for consistent path naming under all of the different supported operating systems. In this example, the string which was written was stored in a variable, so it is impossible to discern what its resulting contents are from just this line.

Example: Writing to a relative path

```
>> fs_write("../mystery/inconspicuous.txt", a_secret)
```

To write a file to a location with a relative path, denote the up directory motion by using a double period '..'. In this example, we are writing the string contained in the variable 'a_secret' to a file named 'inconspicuous.txt', which happens to be one directory up from the current working directory, and from there inside a directory named 'mystery. It is possible to jump up through multiple directories in a single relative path if necessary.

6.3.2.4 help()

```
help( string topic )
```

> **Returns:** *nothing*
>
> The help function may be used to print up various bits of information about the coupling interface and the included modules. Help topics include: general help, functions, modules and functions of modules. When called on a module or for general information, it will include a list of functions provided by said topic.

Example: General help

```
>> help()
```

This will print out a good bit of basic help information for overseer in general. It will produce a list of which modules it was compiled with as well as a list of which functions it provides. To learn more about any of these functions or modules, a separate call to the help function should be made with a string containing the name of the target (omit the trailing parenthesis for functions).

Example: Topical help

```
>> help("melcor.run")


melcor.run( string input_file )


Returns: nothing


This function first executes the MELGEN executable on the specified input file
and then subsequently executes the MELCOR executable on it. The value for
argument 'input_file', should be the location of the input file that MELCOR
should operate on.


Internally, OVR will change directories into the directory containing the input
file, run the executables from there, and then change directories back to where
it was before. This is done so that the output created by the executables will
end up in the same directory as their corresponding input file, which may not
be the same folder in which the OVR script resides, assuming of course that
there was one.
```

This particular example prints out help information on the MELCOR function topic 'melcor.run'. It should be somewhat similar to what is provided by this user guide.

6.3.2.5 print()

```
print( basic_DT data optional:{basic_DT data, ...} )
```

Returns: *nothing*

The print function writes to the standard output, the value(s) of any basic OVR datatype(s). The print function is unique in that it can be called with any number of arguments. If no arguments are specified it will print a newline to the standard output. If multiple arguments are specified it will print them out with a space character in-between the values. It is not required that the data types of the arguments be of the same.

Example: Print with multiple arguments

```
>> print("There are", 1337, "time steps in this solution.")
There are 1337 time steps in this solution.
```

Internally, the non-string arguments of a print function call are converted into strings and then the result is combined into a single string before printed to the standard output.

6.4 Report Module

This module was included as central location for configuration of general OVR post processing capabilities. Since the initial implementation of OVR post processing is limited to the semi-automated generation of plots, the only general configuration currently supplied by this module is for setting up the use of the executable used for plotting, gnuplot.

To fully take advantage of the post-processing capabilities provided by OVR, the system it is being used on should have a copy of gnuplot program installed. Gnuplot is a free and open source graphing utility which can be downloaded from the gnuplot project homepage (`http://www.gnuplot.info`).

6.4.1 Functionality

6.4.1.1 report.set_gnuplot_path()

```
report.set_gnuplot_path( string location )
```

Returns: *nothing*

This function sets the location of gnuplot executable for use with OVR. The value for argument 'location', should be the location of the binary which is to be used by all of the graph functions provided by other modules. Since OVR does not use configuration files, this executable path information must instead be specified by the use of this function. As such, this function is must be run before the user is able to use of the graph functions provided by other modules. The location used with this function should contain both the binary filename as well as path information and must be specified in standard OVR location format.

Example: Setting the gnuplot path

```
gp_path = "C:/gnuplot/binary/gnuplot.exe"
report.set_gnuplot_path(gp_path)
```

In this example, the location supplied to the 'gp_path' variable is supposed to be a valid gnuplot binary. Like other functions which use location strings, the report.set_gnuplot_path function will check to verify that the specified path does exist. If the specified path happens to be incorrect, an error will be produced and reported.

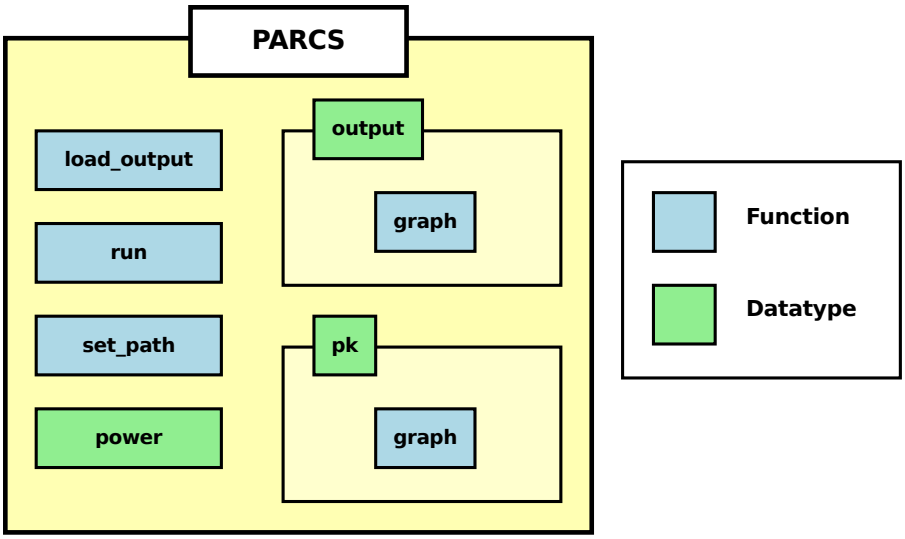

Figure 6.8: A graphical representation of the structure of the PARCS module.

6.5.1 Datatypes

With the inclusion of the PARCS module, the user may encounter a number of additional datatypes, which are specific to the PARCS module. These PARCS module datatypes include: *parcs__output*, *parcs__power*, and *parcs__pk*.

These PARCS module datatypes may be used directly by the functions provided by the PARCS module or potentially by a function of the MELCOR module. Since most of the development effort has been towards accomplishing a coupling from PARCS/AGREE to MELCOR and not the other direction, there isn't as much internal manipulation of the PARCS module datatypes. The melcor.convert_power and melcor.convert_pk functions are the two major coupling related functions which utilize the PARCS module datatypes. While both of these functions operate on the *parcs__output* datatype directly, they technically only do so to access the internally stored *parcs__power* and *parcs__pk* datatypes.

The rest of this section is dedicated to a more detailed explanation of these PARCS module datatypes.

6.5.1.1 parcs__output

The PARCS output datatype contains all of the information extracted from a number of PARCS input/output files. More specifically, the file data that may be stored within this datatype may be from some or all of the following parcs input/output files: "*.inp", "*.map", "*.out", and "*.pkd". The files loaded above must all be from files which have a consistent base name (i.e. the value of the "*" must be equivalent for all of the file extensions listed above).

Currently the only way to pass and to select a spatial power distribution (stored within a *parcs__power* datatype) is handled within the melcor.conver_power function. Since a number of *parcs__power* and a single *parcs__pk* datatype are stored within a single *parcs__output* datatype, it is more convenient to pass the entire object and let the function and let it select the one for the appropriate timestep. So in most cases, this datatype is simply used as a container for the other, more internally used datatypes.

Currently the only way to obtain a PARCS output datatype is by use of the parcs.load_output() function.

The following function(s) can utilize this datatype:

parcs.graph()

```
parcs.graph( parcs__output dt )
```

When used with the *parcs__output* datatype, the parcs.graph() function will by default, generate all plots that it can from the PARCS output file. Since less emphasis was put towards analysis of the PARCS output, this results in only a single plot of the transient reactivity value over the entire range of the PARCS simulation time.

melcor.convert_power()

```
melcor.convert_power( parcs__output dt, maplist )
```

This function will convert from a supported alternate datatype, to a *melcor__power* datatype. When used with a *parcs__output* datatype, a valid *melcor__power* datatype will be returned. Additionally, a maplist must be provided to map the translation between the models. For further explanation about the reason, use, and specification of maplists and powermaps, please refer to the section on core to core power mapping [6.2.1].

6.5.1.2 parcs__pk

The PARCS point kinetics datatype contains information required to describe a transient power event. Currently the only way to obtain a *parcs__pk* datatype is by use of the parcs.load_output() function. Even when this is done, the resulting *parcs__pk* datatype is stored within the *parcs__output* datatype, and so there is no direct way for the user to access it. The data used to populate this datatype is extracted from a PARCS "*.pkd" output file.

Though this datatype is stored away within the *parcs__output* datatype, it does find use whenever the melcor.convert_pk function is used. Further manipulation and preparation of the point kinetics information occurs with the resultant *melcor__pk* datatype. Additionally, this datatype is also capable of producing a plot of its stored transient reactivy. Although this datatype is not directly accessible, this graphing capability is leveraged whenever the parcs.graph function is used on an encompassing *parcs__output* datatype.

The following function(s) can utilize this datatype:

parcs.graph()

```
parcs.graph( parcs__pk dt )
```

When used with the *parcs__pk* datatype, the parcs.graph() function will create a plot of the transient reactivity value over the entire range of the PARCS simulation time. While the user is not currently able to call this function directly on the *parcs__pk* datatype, a call of parcs.graph() on a *parcs__output* datatype does subsequently result in the parcs.graph() function being used on the underlying *parcs__pk* datatype, thus producing the desired behavior.

6.5.1.3 parcs__power

The PARCS power datatype contains spatial power density for a single PARCS timestep. Currently the only way to obtain a *parcs__power* datatype is by use of the parcs.load_output() function. Even when this is done, all of the resulting *parcs__power* datatypes are stored within the *parcs__output* datatype, and so there is no direct way for the user to access them. The data used to populate this datatype is extracted from a PARCS "*.map" output file. While this PARCS datatype is never used directly by the user, the contents of this datatype are used by the melcor.convert_power() function, which has the capability to access the *parcs__power* datatypes stored within a *parcs__output* datatype.

6.5.2 Functionality

With the inclusion of the PARCS module, the user may encounter a number of additional functions, which are specific to the PARCS module. These PARCS module functions include: parcs.graph(), parcs.load_output(), parcs.run(), and parcs.set_path(). The rest of this section is dedicated to a more detailed explanation of these PARCS module functions. Since these functions are provided by the PARCS module, the user must include module (i.e. "parcs.*()") when calling one of these functions.

6.5.2.1 parcs.graph()

```
parcs.graph( parcsDT* datatype, string name, string selection )
```

Returns: *nothing*

This function will produce graphs of the data stored in a PARCS datatype. The value for argument 'datatype', should be the native PARCS datatype that is to be plotted. The value for argument 'name', should short string describing the plot to be created. This short string will become part of the resultant filename for the plot. The final argument 'selection', is used to select which plots are to be created. At this point in time, only the supported value for this argument is "all".

The following datatype(s), can be used with this function:

parcs__output, parcs__pk

Example: Graphing the *parcs__output* datatype

```
parcs.graph(p_out, "RE-trans", "all")
```

In this example, the variable 'p_out' is assumed to be a *parcs__output* datatype. The argument 'name', is set to "RE-trans", this will cause the resulting output file names to be prefixed by this name string. Finally, the 'selection' argument is set to "all", which will result in the production of all possible plots (that the PARCS OVR module is capable of) from this datatype. At this point in time, the only plot which can be produced from a *parcs__output* datatype is for system reactivity versus time.

6.5.2.2 parcs.load_output()

```
parcs.load_output( string location )
```

Returns: *parcs__output*

This function loads some of the output generated from a corresponding PARCS input file into a *parcs__output* datatype. The value for argument 'location', should be the location of the input file that the corresponds to

the desired output files. This location should contain both the file name as well as path information and must be specified in standard OVR location format. In addtion to the information of the specified PARCS input file, this function will attempt to load data from following files associated with the given PARCS input file: "*.pkd", "*.map", and "*.out".

Example: Standard output loading

```
inp_parcs = "c:/somedir/some_deck/steady-state.inp"
output_DT = parcs.load_output(inp_parcs)
```

In this example, the variable 'output_DT', ends up containing a *parcs__output* datatype. The information contained by this variable, comes from the output file corresponding to "steady-state.inp". Notice here that the parcs.load_output function, uses the location of the input file as an argument. It is often the case that the user may want to both run and load the output from a given PARCS run. Since the location of the input file would have to be specified to run PARCS anyhow, it should be convenient for the user to use that same location with the parcs.load_output function. In addition to making sure the correct corresponding output is loaded, this is also done in an effort to allow OVR input scripts to be more concise.

6.5.2.3 parcs.run()

```
parcs.run( string input_file )
```

Returns: *nothing*

This function first will execute PARCS on the specified input file. The value for argument 'input_file', should be the location of the input file that PARCS should operate on.

Internally, OVR will change directories into the directory containing the input file, run the executable from there, and then change directories

back to where it was before. This is done so that the output created by the executable will end up in the same directory as their corresponding input file, which may not be the same folder in which the OVR script resides, assuming of course that there was one.

Example: Starting PARCS using OVR

```
parcs_bin = "C:/parcs/BIN/agree10.exe"
parcs.set_path(parcs_bin)
parcs.run("../parcs_inp/ss-deck.inp")
```

In this example, the parcs.run function is called directly on a file called "ss-deck.inp". It will execute whichever PARCS executable was specified by parcs.set_path on this input file. In this example, a version of AGREE was used started by OVR.

6.5.2.4 parcs.set_path()

```
parcs.set_path( string location )
```

Returns: *nothing*

This function sets the location of PARCS executable for use with OVR. The value for argument 'location', should be the location of the binary which is to be used by the parcs.run function. Since OVR does not use configuration files, executable path information must instead be specified by the use of this function. As such, this function is must be run before the user is able to use the parcs.run function. The location used with this function should contain both the binary filename as well as path information and must be specified in standard OVR location format.

Example: Setting the PARCS path

```
PARCS_exe = "/opt/parcs/bin/agree"
melcor.set_path(PARCS_exe)
```

In this example, the location supplied to the 'PARCS_exe' variable is supposed to be a valid PARCS executable. Like other functions which use location strings, the parcs.set_path function will check to verify that the specified path does exist. If the specified path happens to be incorrect, an error will be produced and reported. One important difference between the parcs.set_path function and the melcor.set_path function is that, the location used for the parcs.set_path function should point directly to a PARCS executable, not the directory which contains it.

6.6 MELCOR Module Layout

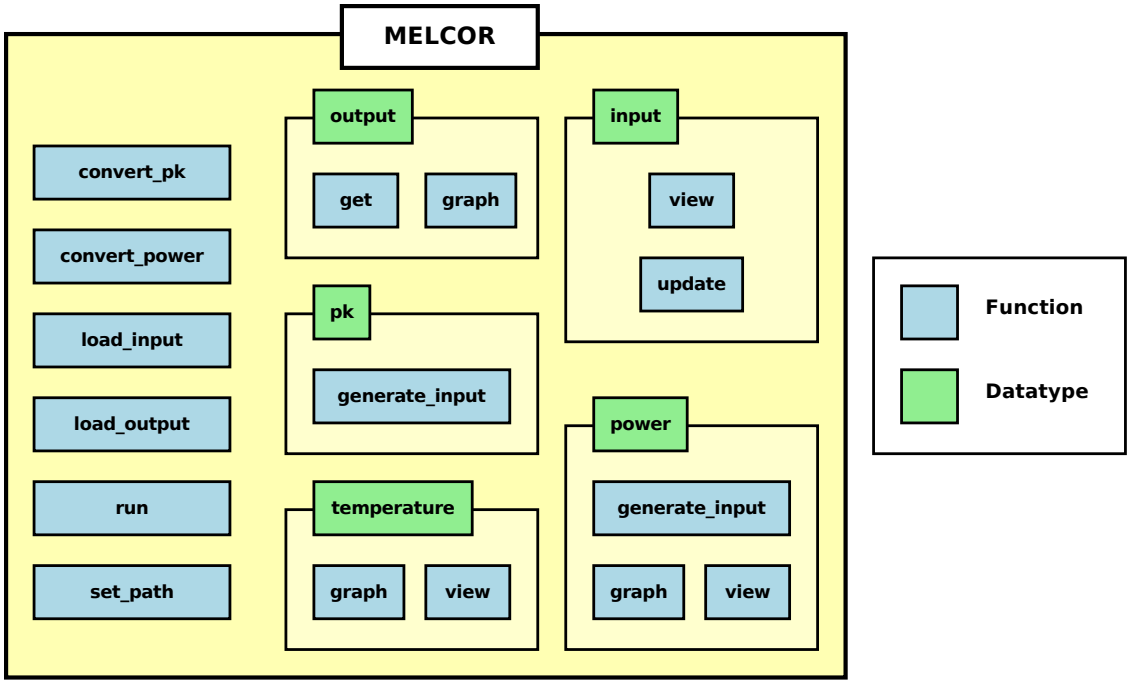

Figure 6.9: A graphical representation of the structure of the MELCOR module.

6.6.1 Module Datatypes

With the inclusion of the MELCOR module, the user may encounter a number of additional datatypes, which are specific to the MELCOR module. These MELCOR module datatypes include: *melcor__input*, *melcor__output*, *melcor__pk*, *melcor__power*,

and *melcor__temperature*. While a number of these additional datatypes may be used in conjunction with other code modules, some of these datatypes only get used internally.

In more detail, the roles that these individual datatypes play when used in a coupling, is as follows. The *melcor__input* datatype may be used with the melcor.update function to quickly insert information generated by ovr into an existing input file. The included *melcor__output* datatype provides only a number of post processing capabilities, namely by use of the melcor.get and melcor.graph functions. The *melcor__pk* datatype is used to produce MELCOR input specifying general point kinetics information as well as transient reactivity. The *melcor__power* datatype is used to produce MELCOR input specifying a spatial power profile. The *melcor__temperature* datatype provides only a number of post processing capabilities, namely by use of the melcor.view and melcor.graph functions.

The rest of this section is dedicated to an even more detailed explanation of these MELCOR module datatypes.

6.6.1.1 melcor__input

The MELCOR input datatype contains all of the information extracted from a MELCOR input file. The usefulness of this datatype with respect to coupling, stems from its capability of working with the melcor.update function. Working with the melcor.update function allows a number of newly generated MELCOR input sections to be merged correctly into the encompassed input file data. Once all of the desired modifications have been made, the resultant contents can then be extracted again by using the melcor.view function.

Currently the only way to obtain a MELCOR input datatype is by use of the melcor.load_input() function.

The following function(s) can utilize this datatype:

melcor.update()

```
melcor.update( melcor__input inp, string text optional:{string flag, ...} )
```

The input supplied by the string 'text' should be of standard MELCOR input format. It is possible to supply multiple input sections within the same input string. If a flag of value, "tab" is included, all sections are expected to be tabular in nature. By default the mode of operation for melcor.update(), is to merge input sections, giving preference to new data entries versus old ones. Alternatively, if a flag of value, "replace" is included, this behaviour changes so that the equivalent sections already existing in the melcor__input datatype are completely overwritten by the ones supplied by the 'text' string.

Note: Currently, melcor.update() only works when used with tabular input sections (those including a "tab" flag). [OVR version 2.771]

melcor.view()

```
melcor.view(melcor__input inp)
```

When the melcor.view() function in used in conjunction with a *melcor__input* datatype, it will return a string which contains the complete input file contents.

6.6.1.2 melcor__output

The MELCOR output datatype contains all of the information extracted from a MELCOR output file.

From a coupling standpoint, this datatype is useful only for quick data post-processing. This post-processing capability is provided by usage of the melcor.get and melcor.graph functions. When used jointly, these functions allow the user to quickly generate customizable plots of individual spatial distributions.

Currently the only way to obtain a MELCOR output datatype is by use of the melcor.load_output() function.

The following function(s) can utilize this datatype:

melcor.get()

```
melcor.get( melcor__output outp, string target optional:{* element} )
```

The melcor.get() function allows the user to access the internally stored datatypes of a MELCOR module dataype.

A number of other MELCOR module datatypes are stored inside of the melcor__output datatype. Should the need arise, these other module datatypes may be accessed individually for any particular simulation time step. The following values for the 'target' string may be used to access the internal datatypes:

- "avg_comp_temps": A list of spatial average component temperatures (as extracted from the output file).
- "avg_fluid_temps": A list of spatial average fluid temperatures (as extracted from the output file).
- "power_densities": A list of control volume powers (as extracted from the output file).

To access a particular element of an internal datatype list, the key corresponding to the element must be exact. To check and see what keys are available for a particular internal datatype, the user may pass a value of "keys" to the optional 'element' argument.

melcor.graph()

```
melcor.graph(melcor__outputDT)
```

When used with the melcor__output datatype, the melcor.graph() function will by default, generate plots of all the information contained. Alternatively, the following

targets may be manually specified by the addition and usage of the 'targets' optional argument with melcor.graph():

- "ftemp": average fluid temperatures,
- "ctemp": average component temperatures,
- "powers": control volume powers,
- "p_of_t": total system power as a function of time,
- "T_max_comp": maximum component temperature as a function of time,

The user is able to plot individual stored items by using the melcor.get() function and then subsequently, the melcor.graph() function on the desired MELCOR module datatype. All other general configuration options provided by the melcor.graph() function are also available for use with this datatype.

6.6.1.3 melcor__pk

The MELCOR point kinetics datatype contains information required to describe a transient power event. When a transient event is to be coupled between PARCS/AGREE and MELCOR, the information stored by this datatype can be used to produce input for MELCOR which specifies this same information. Depending on the types of information are to be coupled, it may be necessary to generate up to three different blocks of MELCOR input. These, three input types are explained below in the melcor.generate_input() subsection for this datatype.

Currently the only way to obtain a MELCOR point kinetics datatype is by use of the melcor.convert_pk() function.

The following function(s) can utilize this datatype:

melcor.generate_input()

```
melcor.generate_input(melcor__pk pktype, string type *... )
```

When used with the melcor__pk datatype, the melcor.generate_input() function requires a string argument referred to here as 'type'. The 'type' argument further specifies the style of input generation from the melcor__pk datatype. The following three values for the 'type' argument are accepted:

- "COR_SC":

  Which, requires the following additional arguments:

  `optional:{int [sections ...]}`

  This operational type generates a MELCOR COR_SC tabular input section. This input is returned as a string. The optional integer list argument 'sections' defines which COR_SC input sections should be generated. Some of these values may vary with time, as such, a time value must be passed so that particular values can be isolated. Currently the following COR_SC sections can be specified: 1404 and 1405.

  Note: Though support is planned, currently COR_SC section 1406 input can not be generated. [OVR version 2.771]

- "reactivity":

  Which, requires the following additional arguments:

  `string cfName, int cfNumber, string edfName`

  This operational type generates MELCOR CF code to bind reactivity to the time dependent value specified by the reactivity EDF. The 'cfName' argument is should be the name of the MELCOR control function being used as the

reactivity value. The 'cfNumber' argument is the CF value to start with (this is required since a number of CFs will need to be produced to bind this value to an EDF). The 'edfName' argument is the file name for the MELCOR EDF that the reactivity value will be bound to.

- "EDF":

  Which, requires the following additional arguments:

  `string variable, optional:{float time_interval}`

  This operational type will generate EDF content containing two values per line (time-value pairs). The 'variable' argument is the particular point kinetics parameter which may be changing as a function of time. Currently the only acceptable value for the 'variable' argument is "reactivity". Optionally, a value may be specified for 'time_interval' which will effectively reduce the number of data points that will end up in the resulting EDF.

While it may appear strange at first that three there are three different operations for the melcor__pk datatype, this is the case because, all three of these methods are required to model a control rod ejection transient while still allowing for a degree of design flexibility for the MELCOR input deck developer.

6.6.1.4 melcor__power

The MELCOR power datatype contains information which describes a spatial power distribution within the reactor core. For any coupling from PARCS/AGREE to MELCOR, if spatial power information is to coupled, it must first be converted into this datatype. After this conversion has taken place, input for MELCOR which specifies the underlying distribution can then be generated. Since there is more than one way for this information to be specified to MELCOR, the melcor.generate_input

function requires that a resulting input style be specified. Additionally, the values of a converted *melor__power* datatype can be visualized by using the melcor.graph function.

Currently the only way to obtain a MELCOR power datatype is by use of the melcor.convert_power() function.

The following function(s) can utilize this datatype:

melcor.generate_input()

```
melcor.generate_input( melcor__power ptype, string type *... )
```

This function generates and returns a string containing MELCOR input sufficient to describe the spatial power distribution contained within the datatype. When used with the melcor__power datatype, the melcor.generate_input() function requires a string argument referred to here as 'type'. The 'type' argument further specifies the style of this input generation from the melcor__power datatype. The following two values for the 'type' argument are accepted:

- "CF_SET":

  This operational type requires no additional arguments.

  This input generation style results in MELCOR input which describes the spatial power distribution by using numerous control functions. In more detail, this is accomplished by the creation of a control function heat source table (COR_QHS) and then subsequent generation of a number of control functions with appropriate names, so as to be referenced by the COR_QHS table. The numbering of the MELCOR control functions starts from the value supplied by the melcor.convert_power() function and increments up from there.

  While this input style should be suitable for a steady state coupling, for analysis of a transient, the "COR_ZP,RP" style should be used instead. While this input

generation style requires less effort to use, this style should in general be treated as a legacy option, with the "COR_ZP,RP" option as the favored alternative.

- "COR_ZP,RP":

  Which, requires the following additional arguments:

```
tab_ZP, tab_RP, melcor__input inp
```

  This input generation style results in MELCOR input which describes the spatial power distribution by using both COR_ZP and COR_RP tables. In more detail, an axial power distribution can be set by using a MELGEN axial level parameters table (COR_ZP). Similarly, a radial power distribution can be set by using a MELGEN radial ring parameters table (COR_ZP).

  While a number of the values required to specify both the COR_ZP and COR_RP tables are automatically calculated by OVR, there remain a number of values for each entry which must be manually specified by the user. These required values are passed into the melcor.generate_input() function by the additional 'tab_ZP' and 'tab_RP' arguments. Further explanation of these arguments is presented below.

  ZP_table sample:

  The contents of the ZP_table are similar to what would be in a MELCOR COR_ZP table, namely [[!Z, PORDP, IHSA], ... , [...]]. For additional information regarding these arguments, please refer to the MELCOR user guide.

```
tab_ZP = [[-4.35, por1, "'COR-RAD-BND-A1'"],
          [ -4.0, por2, "'COR-RAD-BND-A2'"],
          [ -3.0, por2, "'COR-RAD-BND-A3'"],
```

```
                                .
                                .
                                .
                [ 10.5, por1, "'COR-RAD-BND-A27'"],
                [ 11.0, por1, "'COR-RAD-BND-A28'"],
                [ 11.5, por1, "'COR-RAD-BND-A29'"]]
```

RP_table sample:

The contents of the RP_table are similar to what would be in a MELCOR COR_RP table, namely [[!RINGR, IHSR, ICFLHF, ICFCHN, ICFBYP], ..., [...]]. For additional information regarding these arguments, please refer to the MELCOR user guide.

```
tab_RP = [[1.000, "'COR-RAD-BND-A1'", "NO", "NO",       "NO"],
          [1.170, "'COR-RAD-BND-A2'", "NO", "'FLDIR2'", "'FLDIR2'"],
                                      .
                                      .
                                      .
          [2.436, "'COR-RAD-BND-A7'", "NO", "NO",       "NO"],
          [2.606, "'COR-RAD-BND-A8'", "NO", "NO",       "NO"]]
```

melcor.graph()

```
melcor.graph(melcor__powerDT optional:{* string filename} )
```

When used with the melcor__power datatype, the melcor.graph() function will by default, generate a 3d spatial power plot which covers the entire range of radial rings and axial levels as specified by the MELCOR input file. Unlike some of the other MELCOR datatypes, the style of the resultant plot from this datatype may not be configured by the use of general optional arguments to graph(). However, the resultant filename for the plot may be specified by the addition of a single string

argument.

melcor.view()

```
melcor.view(melcor__powerDT, string format)
```

This function creates and returns human readable output visualizing the data inside of a *melcor__power* datatype, in the form of a string. The following two values for the 'format' argument are accepted:

- "x_rings":

  This generates an output which begins with line of description followed by lines of data. This format has the x-axis as radial rings and y-axis as axial levels.

- "y_rings":

  This generates an output which begins with line of description followed by lines of data. This format has the y-axis as radial rings and x-axis as axial levels.

6.6.1.5 melcor__temperature

The MELCOR temperature datatype contains information which describes a spatial temperature distribution within a MELCOR model. From a coupling standpoint, this datatype is only available for post-processing techniques. Associated functionality is provided to both directly generate customizable plots of contained data as well as, to generate output that can be easily loaded into most spreadsheet programs.

Currently, the only way to access a MELCOR temperature datatype directly is, by extraction from a melcor__output datatype by use of the melcor.get() function.

The following function(s) can utilize this datatype:

melcor.graph()

```
melcor.graph(melcor__temperatureDT)
```

When used with the melcor__temperature datatype, the melcor.graph() function will by default, generate a 3d spatial temperature plot which covers the entire range of radial rings and axial levels as specified by the MELCOR input file.

With the exception of the 'targets' optional argument, all other general configuration options provided by the melcor.graph() function are available for use with this datatype.

melcor.view()

```
melcor.view(melcor__temperatureDT, string format)
```

This function creates and returns human readable output visualizing the data inside of a *melcor__temperature* datatype, in the form of a string. The following two values for the 'format' argument are accepted:

- "x_rings":

  This generates an output which begins with line of description followed by lines of data. This format has the x-axis as radial rings and y-axis as axial levels.

- "y_rings":

  This generates an output which begins with line of description followed by lines of data. This format has the y-axis as radial rings and x-axis as axial levels.

6.6.2 Functionality

With the inclusion of the MELCOR module, the user may encounter a number of additional functions, which are specific to the MELCOR module. These MELCOR module functions include: melcor.convert_pk(), melcor.convert_power(), melcor.generate_input(), melcor.get(), melcor.graph(), melcor.load_input(), melcor.load_output(), melcor.run(), melcor.set_path(), melcor.update(), and melcor.view(). The rest of this section is dedicated to a more detailed explanation of

these MELCOR module functions. Since these functions are provided by the MELCOR module, the user must include module (i.e. "melcor.*()") when calling one of these functions.

6.6.2.1 melcor.convert_pk()

```
melcor.convert_pk( modDT* dt )
```

Returns: *melcor__pk*

This function will convert from a supported alternate datatype, to a *melcor__pk* datatype. The value for the 'dt' argument must be one of the supported datatypes.

The following datatype(s), can be used with this function:

parcs__output

6.6.2.2 melcor.convert_power()

```
melcor.convert_power( modDT* dt, maplist )
```

Returns: *melcor__power*

This function will convert from a supported alternate datatype, to a *melcor__power* datatype. The value for the 'dt' argument must be one of the supported datatypes. Additionally, a maplist must be provided to map the translation between the models. For further explanation about the reason, use, and specification of maplists and powermaps, please refer to the section on core to core power mapping [6.2.1].

The following datatype(s), can be used with this function:

parcs__output

Example: Single Block Conversion

```
test_map = [(0.0, (6,25),(10,4)), (2,6,400)]
test_maplist = [test_map]
conv_power = melcor.convert_power(some_output, test_maplist)
```

In this example, a variable named ‘test_map‘ is created. This variable ‘test_map‘, meets the specification of a powermap. Next this variable is placed into a list variable named ‘test_maplist‘. This variable ‘test_maplist‘, meets the specification of a maplist, which is simply a list of power maps. In this particular case, it only contains one power map, ‘test_map‘. The variable ‘some_output‘ is presumed to be a *parcs__output* datatype, which is required for this conversion to take place. The end result of all this would be a *melcor__power* datatype is returned from the function and set as the value of the variable ‘conv_power‘.

6.6.2.3 melcor.generate_input()

```
melcor.generate_input( melcorDT* datatype, *... )
```

Returns: *string*

This function generates input corresponding to the MELCOR datatype it was supplied with. The input that is created is in the form of a string, which allows the user to use it in a number of different ways. The ‘*...’ argument shown here, indicates that further arguments are required, but that they are specific to the datatype that the function is being used with. For more information regarding which additional arguments are required for a particular datatype, please see the melcor.generate_input() subsection which should exist under the section dedicated to explaining that particular datatype.

The following datatype(s), can be used with this function:

melcor__pk, *melcor__power*

Example: Generating input from a *melcor__power* datatype

```
melcor.generate_input(mel_power, "COR_ZP,RP")
```

In this example, mel_power is a *melcor__power* datatype. The additional argument supplied here is for input style, which was given a value of "COR_ZP,RP". This particular datatype can utilize two different values for input style. The differences between these two different input styles are the format of the resultant input, as well as, the number of arguments each requires to be used. As a general rule, to better understand how a particular function will interact with a native datatype, see the datatype specific section regarding that particular function (e.g. in section [6.6.1.4])

6.6.2.4 melcor.get()

```
melcor.get( melcorDT* datatype, string target  optional:{element} )
```

Returns: **something**

This function allows the user to access values from the internal parts of a MELCOR native datatype structure. The value for argument 'datatype', should be the native MELCOR datatype from which a value should be extracted. The value for argument 'target', should be the name of the internal data structure that is to be extracted. Optionally, if the internal data structure is a list and the goal is to grab a single item from it, the 'element' argument must be supplied with an appropriate indexing value. To help the user identify valid index values, the melcor.get() function can also be called on a particular internal data structure with a string value of "keys" for the 'element' argument. This will return to the user a string containing a list of all the available index values in the list.

It must be noted that, the correct usage of this function requires that the user have some knowledge of the internal structure of the native datatypes. While this function technically works on all of the native MELCOR datatypes, at this point in time, only the names of a few of the internal data structures of the *melcor__output* datatype have been disclosed for use. The return value of this function is listed as **something** because, the return value depends entirely on what internal structure the user chooses to access.

The following datatype(s), can be used with this function:

melcor__output

6.6.2.5 melcor.graph()

```
melcor.graph( melcorDT* datatype, optional:{OPT_ARG, ...} )
```

Returns: *nothing*

This function produces one or more graphs from the data within in a MELCOR datatype. The value for argument 'datatype', should be the native MELCOR datatype that is to be plotted. The value for arguments listed as 'OPT_ARG', depend on the particular plot specification that is to be made. Note that it is possible to specify any number of these general graph arguments for any particular call to melcor.graph().

While not all of the MELCOR datatypes will respond to all of the possible general graph arguments, an exhaustive list of these optional arguments is presented below.

The following datatype(s), can be used with this function:

melcor__output, melcor__power, melcor__temperature

A list of general arguments for melcor.graph():

- "name":

  Which has a structure like the following:

  ```
  ["name", string imgname]
  ```

  This argument allows the user to configure the name of the plot that is to be created. If omitted, a somewhat descriptive and unique name should be generated for any particular plot or set of plots. The value of the argument 'imgname' should be the name of the desired output plot.

- "overwrite":

  Which has a structure like the following:

  ```
  ["overwrite", int auto]
  ```

  This argument allows the user to change the behavior of OVR with regard to the overwriting of existing plots. By default, the user will be prompted about overwriting if a new plot would be generated with the same name as one already existing in the current working directory. The enable unprompted overwriting, the value of the argument 'auto' should set to 1.

- "rings":

  Which has a structure like the following:

  ```
  ["rings", int rmin, int rmax]
  ```

  This argument allows the user to modify the radial ring range that is to be plotted. The default behavior is for a plot to include all of the radial rings that are in the model. The value of the argument 'rmin' should be the new

minimum value for the ring range. The value of the argument 'rmax' should be the new maximum value for the ring range.

- "levels":

  Which has a structure like the following:

  ```
  ["levels", int lmin, int lmax]
  ```

  This argument allows the user to modify the axial level range that is to be plotted. The default behavior is for a plot to include all of the axial levels that are in the model. The value of the argument 'lmin' should be the new minimum value for the level range. The value of the argument 'lmax' should be the new maximum value for the level range.

- "style":

  Which has a structure like the following:

  ```
  ["style", string pstyle]
  ```

  This argument allows the user to change the style of a spatial plot. The default style for these spatial plots is to produce a three dimensional plot. Alternatively, a two-dimensional plot can be generated by setting the value of the argument 'pstyle' to "map".

- "targets":

  Which has a structure like the following:

  ```
  ["targets", string tgt, ...]
  ```

This argument which only works with the *melcor__output* datatype, allows the user to select which groups of output information will be plotted. The default behavior with a *melcor__output* datatype is to produce all possible plots with all other unspecified options set to default values. To reduce the number of plots produced by a *melcor__output* datatype, the user should include as arguments here, a number of strings including only the groups of plots that are desired. For a list of these plot groups, please see the melcor.graph() subsection under the the *melcor__output* datatype section [6.6.1.2].

Example: Graphing the *melcor__output* datatype

```
melcor.graph(mel_output, ["name", "ss-test"], ["targets", "ctemp", "ftemp"])
```

In this example, mel_output is assumed to be a *melcor__output* datatype. The argument 'name', is set to "ss-test", this will cause the resulting output file names to be prefixed by this name string. This sort of prefix naming behavior is unique to the *melcor__output* datatype, as is the only datatype of the MELCOR module that is capable of producing multiple plots. The 'targets' argument is set to "ctemps" and "ftemps", which will result in all of the fluid and component spatial temperature distributions for all output sections being plotted.

6.6.2.6 melcor.load_input()

```
melcor.load_input( string location )
```

Returns: *melcor__input*

This function loads the data from a MELCOR input file into a *melcor__input* datatype. The value for argument 'location', should be the location of the input file that is to be loaded. This location should contain both the file name as well as path information and must be specified in standard OVR location format.

Example: Loading an input file

```
mel_inp_file = "c:/somedir/some_deck/steady-state.inp"
the_input = melcor.load_input(mel_inp_file)
```

In this example, the variable 'the_input', ends up containing a *melcor__input* dataype. The information contained by this variable, comes from the MELCOR input file "steady-state.inp". This function is most often used in situations where the user wishes to modify an existing input file.

6.6.2.7 melcor.load_output()

```
melcor.load_output( string location )
```

Returns: *melcor__output*

This function loads some of the output generated from a corresponding MELCOR input file into a *melcor__output* datatype. The value for argument 'location', should be the location of the input file that the corresponds to the desired output file. This location should contain both the file name as well as path information and must be specified in standard OVR location format.

Example: Standard output loading

```
mel_inp_file = "c:/somedir/some_deck/steady-state.inp"
some_output = melcor.load_output(mel_inp_file)
```

In this example, the variable 'some_output', ends up containing a *melcor__output* dataype. The information contained by this variable, comes from the output file corresponding to "steady-state.inp". Notice here that the melcor.load_output function, uses the location of the input file as an argument. It is often the case that the user may want to both run and load the output from a given MELCOR run.

Since the location of the input file would have to be specified to run MELCOR anyhow, it should be convenient for the user to use that same location with the melcor.load_output function. In addition to making sure the correct corresponding output is loaded, this is also done in an effort to allow OVR input scripts to be more concise.

6.6.2.8 melcor.run()

```
melcor.run( string input_file )
```

Returns: *nothing*

This function first executes the MELGEN executable on the specified input file and then subsequently executes the MELCOR executable on it. The value for argument 'input_file', should be the location of the input file that MELCOR should operate on.

Internally, OVR will change directories into the directory containing the input file, run the executables from there, and then change directories back to where it was before. This is done so that the output created by the executables will end up in the same directory as their corresponding input file, which may not be the same folder in which the OVR script resides, assuming of course that there was one.

Example: Starting MELCOR using OVR

```
melcor.run("../mel_inp/ss-deck.inp")
```

In this example, the melcor.run function is called directly on a file called "ss-deck.inp". It will execute MELGEN and MELCOR on this input file and in that order. Depending on input file construction as well as the presence of preexisting output files, these programs may require some additional user input prior when starting up (e.g. MELCOR may ask if the user wants to overwrite existing output files). Since, OVR can

not automatically enter this information, the user may need to monitor the progress of these programs, at least until they have started successfully.

6.6.2.9 melcor.set_path()

```
melcor.set_path( string melpath )
```

Returns: *nothing*

This function sets the location of MELCOR executables for use with OVR. The value for argument 'melpath', should be the location of the directory which contains both a pair of valid MELCOR and MELGEN binaries. Since OVR does not use configuration files, executable path information must instead be specified by the use of this function. As such, this function is must be run before the user is able to use the melcor.run function. The location used with this function should contain only the path to the directory containing the binary files (including a final path separator character "/") and must be specified in standard OVR location format.

Example: Setting the MELCOR path

```
mel_path = "c:/melcor/melcor_executables/PC/windows/optimized/"
melcor.set_path(mel_path)
```

In this example, the location supplied to the 'mel_path' variable is assumed to contain both the "melcor.exe" and "melgen.exe" executables. Like other functions which use location strings, the melcor.set_path function will check to verify that the specified path does exist. If the specified path happens to be incorrect, an error will be produced and reported.

6.6.2.10 melcor.update()

```
melcor.update( melcorDT* dt, string newdata optional:{string flag, ...} )
```

Returns: *melcorDT**

This function modifies the contents of the supplied datatype by updating particular data sections. The value for argument 'dt', should be the native MELCOR datatype that is to be modified. The value for argument 'newdata', must be the string containing the new sections to be added. The determination of the data sections to be updated is handled automatically; this information is extracted from the 'newdata' argument. Finally, a number of additional string argument flags may be specified to change the behavior of the update function. These optional string argument flags are explained below in more detail.

The following datatype(s), can be used with this function:

melcor__input

Optional argument flags:

- ***"tab"***:

  When this flag is present, the update function will operate expecting tabular style input from the 'newdata' argument. As a result of this, all string input is expected to be in tabular form when the "tab" flag is present.

- ***"replace"***:

  When this flag is present, the behavior of the update function is changed so that, old data sections will be completely overwritten by the new sections that are present in the 'newdata' argument. The default behavior of the update

function is to merge information of the target with those from the 'newdata' argument. More specifically, this means that any new unique elements of a preexisting section are added anew, and preexisting elements within a section being updated will be replaced by the new values.

Example: Updating an input section

```
mel_pk  = melcor.convert_pk(parcs_output)
pow_str = melcor.generate_input(mel_pk, [1404,1405], 0.0)
new_inp = melcor.update(mel_input, mel_pk_str, "tab")
```

In this example, some newly generated point kinetics information from PARCS is merged with the information contained in the 'mel_input' variable. The input that is created from the melcor.generate_input call produces some MELCOR COR_SC input. Since the COR_SC input section often contains other information besides point kinetics information, the default behavior should be used. The default behavior of update will only merge and update relevant portions of this section and leave the preexisting unrelated entries unharmed. Also, not that since COR_SC is tabular in nature, the "tab" flag must also be set.

Example: Replacing some input sections

```
mel_input = melcor.load_input(inp_path)
pow_str = melcor.generate_input(mel_power, "COR_ZP,RP", ZP_tab,
                                RP_tab, inp_path)
new_inp = melcor.update(mel_input, pow_str, "tab", "replace")
```

In this example, the melcor.update function is being used to update a *melcor__input* datatype stored in the 'mel_input' variable. The value used for the 'type' argument in melcor.generate_input is "COR_ZP, RP". The resulting input from this will contain data for two MELCOR data sections: COR_ZP and COR_RP. This input is also tabular in nature and so the "tab" flag must also be set. Unlike in the previous example, this case requires that no old data should remain from these original input

sections, and so the "replace" flag is set. One important thing to note is that these sections do have to exist in the input file prior to using melcor.update. In the event the input file did not contain these sections, the update would not be able to take place, as the update function would not know where to place the data for those sections.

6.6.2.11 melcor.view()

```
melcor.view( melcorDT* dt, ... )
```

Returns: *string*

This function produces a string representation of the supplied native MELCOR datatype. The value for argument 'dt', should be the native MELCOR datatype that is to be viewed. There can potentially be some addtional arguments required, depending on which dataype the function is being used with. To ensure proper usage of the melcor.view function, it is suggested that the melcor.view() subsection of the desired MELCOR datatype be reviewed.

The following datatype(s), can be used with this function:

melcor__input, melcor__power, melcor__temperature

Example: Viewing a *melcor__power* datatype

```
power_str = melcor.view(mel_power)
print(power_str)
```

In this example, a string representation of the *melcor__power* datatype stored in variable 'mel_power' is created. Since no additional arguments were supplied, the entire power distribution will be included in the view (which is the default behavior of melcor.view with the *melcor__power*). Once this string is created, it can then be however the user sees fit.

7. OVERSEER DEMONSTRATION

To demonstrate the coupling code that was developed, a couple of test problems were selected for further analysis. These two test problems were selected from the OECD/NEA PBMR400 benchmark, because it is both well defined and it would allow for comparison of this code coupling with the results of the other simulation efforts. The first demonstration is of a coupled steady state run between PARCS/AGREE and MELCOR. The second demonstration is of a coupled reactivity insertion transient due to a total control rod ejection.

This chapter of the thesis was designed to accomplish a number of different goals. First, a couple of complete examples are provided here to show the entire process of these couplings. For each example, there is a relatively detailed explanation of what is happening at various parts the coupling script. Finally, for each example there is an included discussion of the output generated by the resultant coupling.

7.1 Coupled PBMR400 Steady-state Simulation

The first example is of a coupled PBMR400 steady state simulation. In this demonstration a PARCS/AGREE model is run in a steady state manner for a few time steps and then stopped. After that simulation completes, the spatial power distribution from the first time step of the PARCS/AGREE run is extracted and used in the MELCOR input. Before this information can be used by MELCOR it must be converted to be in the style that MELCOR expects. Finally, the coupled example is completed by running MELCOR with this converted information.

The following section is in its entirety, the ovr script used to accomplish the described PBMR400 steady state coupling from PARCS/AGREE to MELCOR. The script is structured into a number of logical sections with comments enumerating

and describing the sections Immediately following this, there is a more detailed explanation of each of these input sections.

7.1.1 Script for the PBMR400 Steady-state Coupling

```
## 1 --- set executable paths first
parcs_path = "C:/mleimon/parcs/BIN/agree10.exe"
parcs.set_path(parcs_path)

mel_path = "C:/mleimon/melcor/melcor_executables/PC/windows/optimized"
melcor.set_path(mel_path)

## 2 --- now enter input file locations
parcs_inp = "input--parcs/inp/P400ss.inp"
mel_inp   = "input--melcor/pbmr400.inp"

## 3 --- execute parcs on input file
parcs.run(parcs_inp)

## 4 --- load output from parcs execution
output_parcs = parcs.load_output(parcs_inp)

## 5 --- describe the power mapping from parcs to melcor
pwr_map = [(0.0, (6,25), (10,4)), (2, 6, 400)]
maplist = [pwr_map]

## 6 --- use this maplist (which contains one map) to convert the power
mel_power = melcor.convert_power(output_parcs, maplist)

## 7 --- convert the internal melcor power datatype to a usable string input
mel_pow_str = melcor.generate_input(mel_power, "CF_SET")
```

```
## 8 --- now we will write this power string to a file
power_file = "input--melcor/pwr-source.inp"
fs_write(power_file, mel_pow_str, 1)

## 9 --- now it is time to execute melcor
melcor.run(mel_inp)
```

7.1.2 Detailed Explanation of the Script

The following is a more detailed explanation of the various sections of the ovr script used for the steady-state coupling. These code sections will be referred to by their number, which can be seen at various points in the code (e.g. '`## 3 --- ...`', marks the start of the third code section).

The first two sections of this script define a number of paths and serve as the general setup for the problem. Section one of the script creates two strings: 'parcs_path' and 'mel_path' and then uses them as arguments to the *set_path* functions for both the PARCS and MELCOR modules respectively. This is done so that the interface understands the location of these executables. Section two of the script creates a couple of location strings which are then later used with run functions of the PARCS and MELCOR modules. While it is not necessary that these variables be created before calling the run functions, it is done here to maximize readability and also to allow easy reuse.

Together, the third and fourth sections run PARCS on the specified input file and subsequently load the output that is generated. The line denoting section three executes the version of PARCS previously specified by *parcs.set_path()* on the supplied argument, in this case 'parcs_inp'. In section four the command *parcs.load_output()* loads the output files corresponding to a corresponding PARCS input file, which is what is specified as the argument to this function. This function returns a

parcs__output datatype and so the variable 'output_parcs' is assigned this value.

The fifth and sixth sections of the script both describe an equivalent core-to-core power mapping and then use this mapping to convert a PARCS spatial power distribution to one usable by MELCOR. The fifth section creates a maplist (a list of powermaps), which describes the power information mapping from a PARCS input to a corresponding MELCOR input. The first argument of the 'pwr_map' is (0.0, (6,25), (10,4)), this is information about the PARCS output. More specifically, the first value is the PARCS simulation time of the desired power information, the following two values describe two locations of the PARCS nodalization, a starting and ending point, respectively. The second value (2, 6, 400), is information about the MELCOR input. More specifically, the starting ring, starting level and starting control function number. In the sixth section *melcor.convert_power()* uses a parcs__output datatype in conjunction with a maplist to create a corresponding melcor__power datatype. This function returns a melcor__power datatype, thus the variable 'mel_power' is of this type.

The seventh and eight sections generate MELCOR input describing the spatial power distribution and then write this information into a new input file. In section seven, the function *melcor.generate_input()* is used by supplying it with a *melcor__power* datatype as well as a string "CF_SET", which specifies the style of input generation. Unfortunately, while simpler than other options, the "CF_SET" power specification style is only suitable for steady state calculations. This function returns a string representing a section of valid MELCOR input, this value is assigned to a new variable named 'mel_pow_str'. Within section eight, a variable containing a location string 'power_file' is created and then used by the function *fs_write()* to write the contents of the string in variable 'mel_pow_str' to a file at location 'power_file'. The last argument, '1', is an optional argument which specifies

the unprompted overwriting of preexisting files at the specified location, alternatively the optional argument string "overwrite" accomplishes the same.

Finally, the last section includes a single function which executes the MELCOR executable. More specifically, it first runs MELGEN executable on the supplied input file and then subsequently, the MELCOR executable. This effectively completes the steady-state coupling procedure.

7.1.3 Results and Analysis

Presented here are a number of plots generated from the output of the coupled steady state simulation. Since in this example, only the spatial power information from PARCS was coupled into MELCOR, the focus of this analysis will be on the resultant spatial temperature profiles in the core of the MELCOR simulation. This particular MELCOR input was designed to run with no transient events for over 1000 seconds of simulation time. This long duration of steady state operation was used to allow the solution to stabilize before any results were taken.

This first plot shows the coupled MELCOR steady-state model reaching an equilibrium for the maximum component temperature. Notice that, after about 500-600 seconds, the rate at which this value changes greatly diminishes. An extra data point at 10000 seconds was included to further illustrate how slowly changes occur after this initial ramp up time.

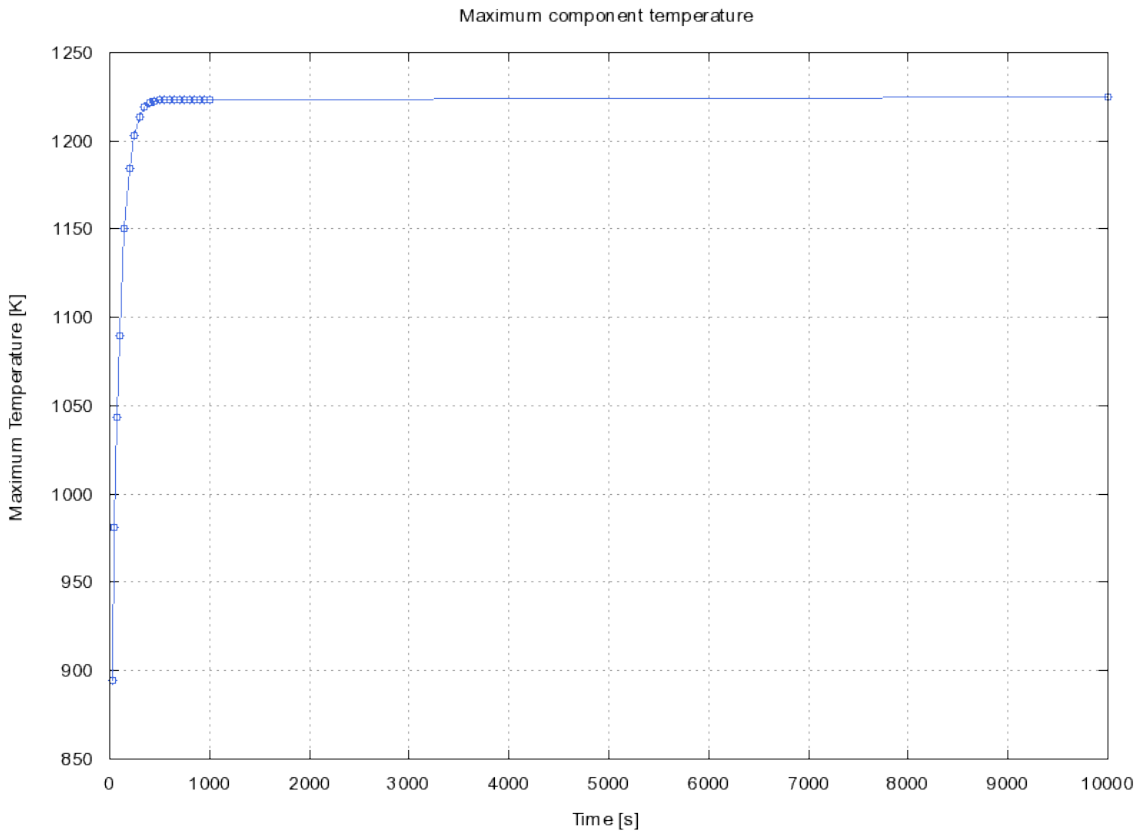

Figure 7.1: A plot of the maximum component temperatures for a steady state simulation. This graphic demonstrates exactly why the simulation is run for around 1000 seconds before the spatial plots are generated.

This figure is of some great importance because the spatial plots for the steady state case were generated from the distributions at a time of 1000 seconds. However, just to make extra certain that the model had completely stabilized first, the transient model in the next section does not start the transient until after 9000 seconds of

simulation time.

The following plot is one of the only included spatial temperature plots which shows the values at all parts of the nodalization. Including all of this information is not particularly useful as the MELCOR model was not designed to have temperatures taken at these locations outside of the core. This is especially true about the upper plenum area (located around axial level 27).

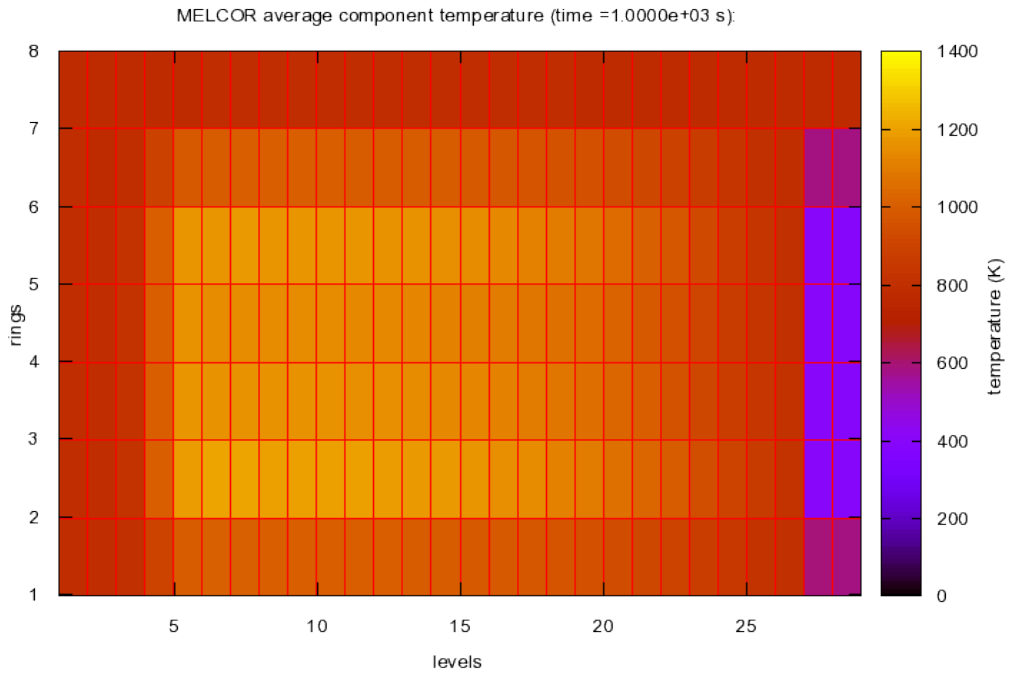

Figure 7.2: A plot of the average spatial component temperatures for a steady state simulation. This distribution came from a MELCOR output file from a simulation time of 1000s.

Even beyond that particular detail, there is another significant reason for the focus on the core region. The reason the focus is on the core region is because that is the only portion of the MELCOR model which is equivalent in nodalization with the

PARCS model. Additionally, it is because of this equivalent core nodalization that these values can be compared with the OECD/NEA PBMR400 benchmark results.

The following figure is a subsection of the previous data set which only includes the core region. It can be seen that with this reduction of area, the z-axis is able to utilize a more detailed range of values in the areas of interest. The maximum temperature value shown in this plot is approximately 1223K.

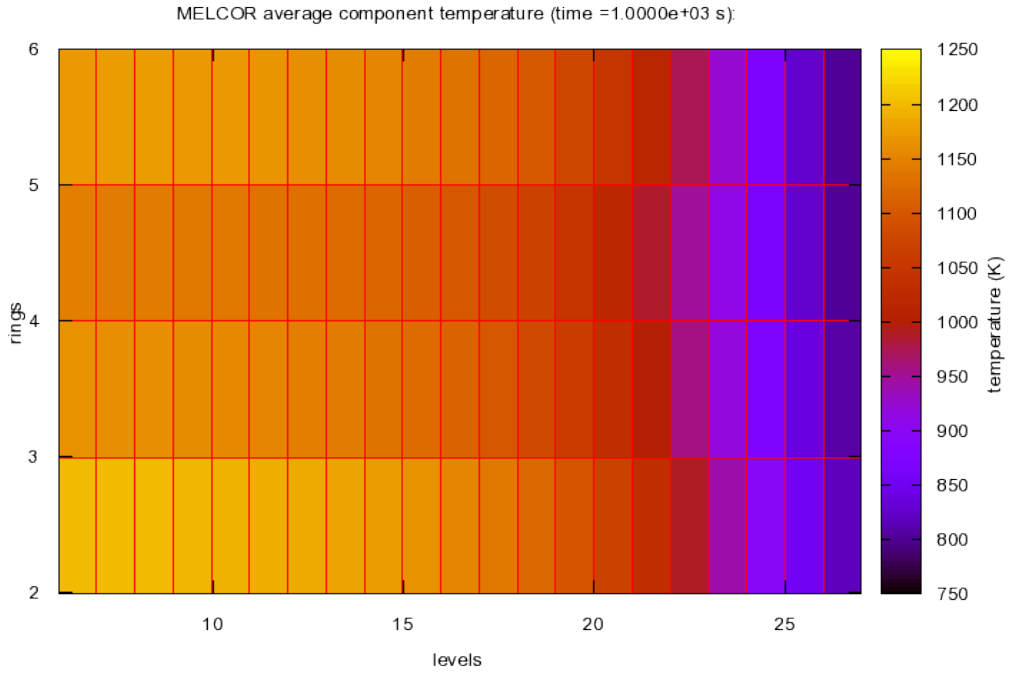

Figure 7.3: A plot of the average spatial component temperatures for a steady state simulation. This distribution came from a MELCOR output file from a simulation time of 1000s. This particular plot includes only the core region of the MELCOR PBMR400 model.

Now, there are some general features of this plot which should be explained. Radial ring 2 of the MELCOR model is the innermost radial ring which contains fuel.

Radial ring 6 of the MELCOR model is the outermost radial ring which contains fuel.

The following figure is of the fluid spatial temperature distribution. Notice that the Helium coolant flows from the top of the core down through the bottom (from right to left in this diagram). Another aspect to notice is how the average temperature of the fluid lags behind that of the average component temperature (as shown in the previous figure). This has to be the case for the fluid temperature to increase as the it flows through the nodes containing said components.

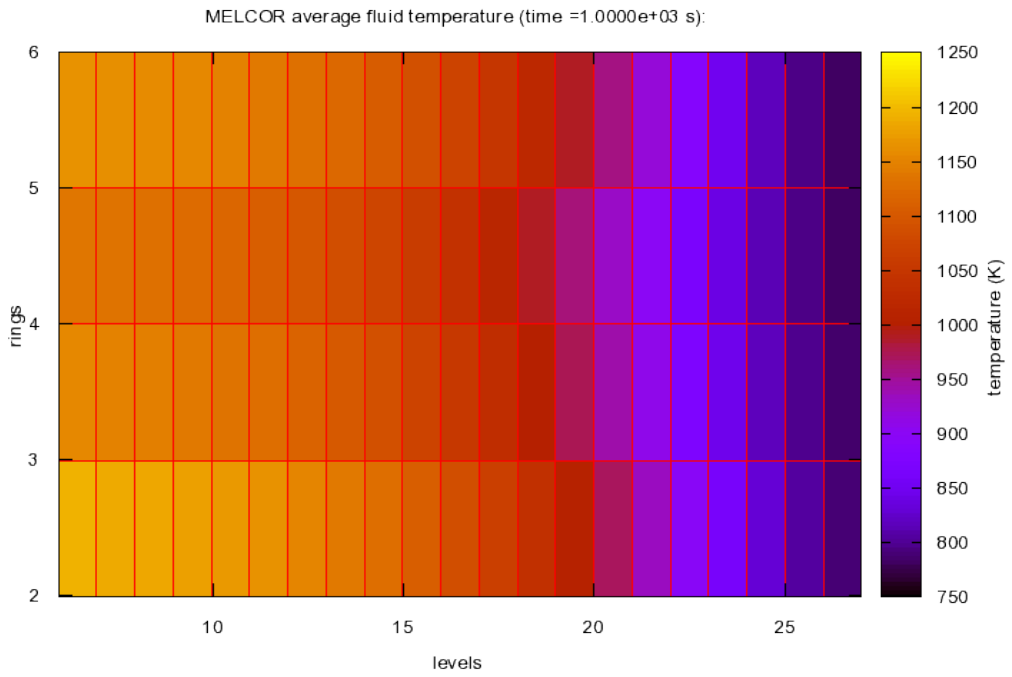

Figure 7.4: A plot of the average spatial fluid temperatures for a steady state simulation. This distribution came from a MELCOR output file from a simulation time of 1000s. This particular plot includes only the core region of the MELCOR PBMR400 model.

Similar to the spatial component temperature distribution, the maximum tem-

perature value also appears here at the bottom of the innermost fuel region of the core (i.e. radial ring 2, axial level 6).

The following table is one of the tabular entries created from this steady state coupling for the OECD/NEA PBMR400 benchmark. While table 7.1 precisely expects fuel temperatures, the closest bit of spatial temperature information that was available from the output was the average spatial component temperature.

	Fuel Temperatures (°C)					
MELCOR-PARCS	*108.5*	*125.5*	*142.5*	*159.5*	*176.5*	ave
25	534.54	525.58	521.79	518.77	520.87	524.31
75	566.94	549.8	542.28	536.33	539.95	547.06
125	610.21	584.18	573.12	564.96	574.3	581.35
175	661.6	627.25	614.29	607.78	638.2	629.82
225	714.18	672.71	657.84	652.52	699.91	679.43
275	761.9	714.96	697.9	692.66	751.2	723.72
325	802.45	751.88	732.79	727.31	793.45	761.58
375	835.66	782.97	762.17	756.37	827.43	792.92
425	862.28	808.53	786.33	780.2	854.2	818.31
475	883.27	829.19	805.89	799.44	874.95	838.55
525	899.67	845.71	821.53	814.83	890.83	854.51
575	912.44	858.86	833.96	827.04	902.87	867.03
625	922.36	869.31	843.83	836.73	911.93	876.83
675	930.08	877.64	851.68	844.43	918.71	884.51
725	936.08	884.27	857.93	850.56	923.73	890.51
775	940.72	889.55	862.9	855.43	927.38	895.20
825	944.26	893.73	866.83	859.3	929.92	898.81
875	946.88	896.99	869.89	862.31	931.54	901.52
925	948.69	899.44	872.19	864.58	932.36	903.45
975	949.79	901.17	873.81	866.19	932.45	904.68
1025	950.21	902.25	874.82	867.18	931.82	905.26
1075	950.52	903.92	876.61	868.51	927.04	905.32
ave	839.31	794.09	772.74	766.07	824.32	

Table 7.1: The data section for an entry to the OECD/NEA PBMR400 steady state benchmark. This data is of average component temperatures as extracted from MELCOR, with the only modification being a change of units from Kelvin to Celcius. The values listed in yellow (horizontally and vertically) are average values for the radial rings and axial levels, respectively.

Internally MELCOR appears to track the mass, volume and heat capacity of both a given fluid and component of every control volume. However, the result of this treatment is that both the fuel and graphite moderator within these control volumes are homogenized and treated as one single composite "component", with regards to temperature information. Because of this treatment, it is not possible to extract precise temperature information about only the fuel or only the moderator by any trivial means. However, for sake of comparison with the OECD/NEA benchmark these "composite" temperatures are used in both fuel and moderator cases, since it is derived from the properties of both.

The following plot is a graphical comparison of the coupled MELCOR-PARCS/AGREE to the other codes from the OECD/NEA PBMR400 benchmark.

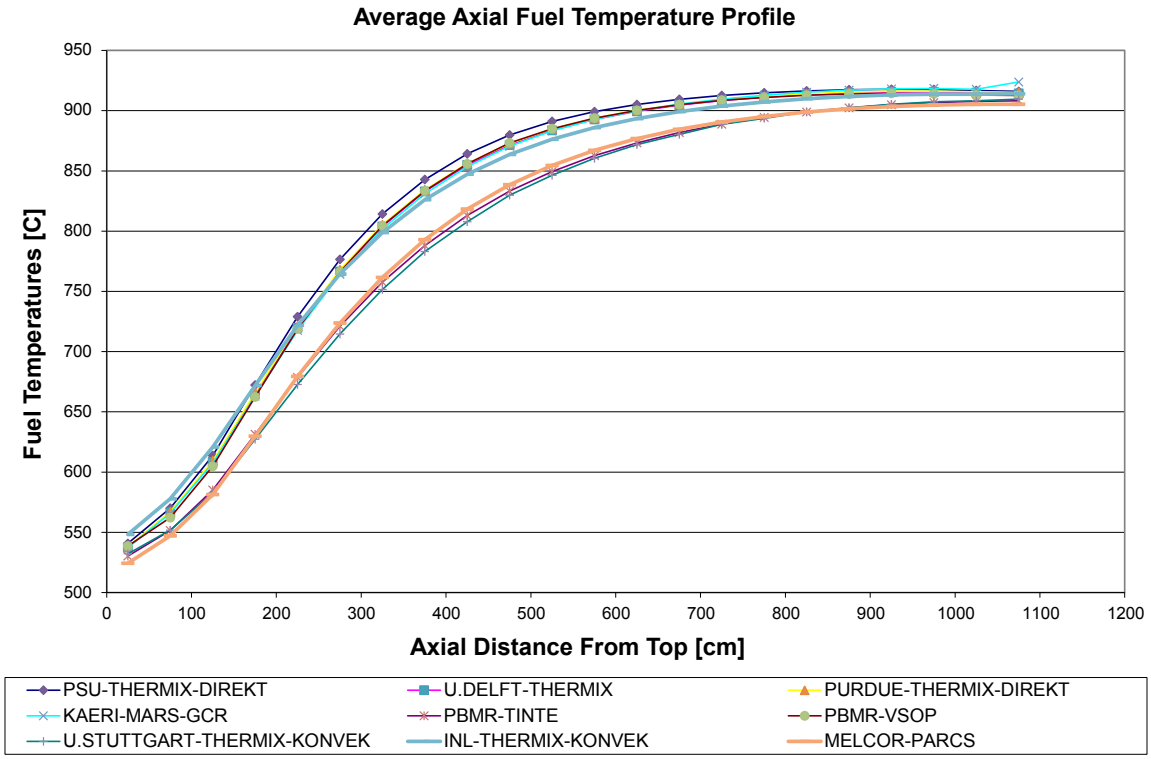

Figure 7.5: A comparison of computed profiles for average axial fuel temperature for the PBMR400 steady state benchmark. The values produced by this research are identified in the figure under the name 'MELCOR-PARCS'.

It can be seen here that the average axial fuel temperature profile, produced from the component temperatures seem to be quite comparable to the fuel temperature profiles of the other codes that were tested in the benchmark.

The following plot is yet again, another graphical comparison of the coupled MELCOR-PARCS/AGREE to the other codes from the OECD/NEA PBMR400 benchmark. Instead in this plot, the axial temperature profile of the moderator (graphite) in the core is shown.

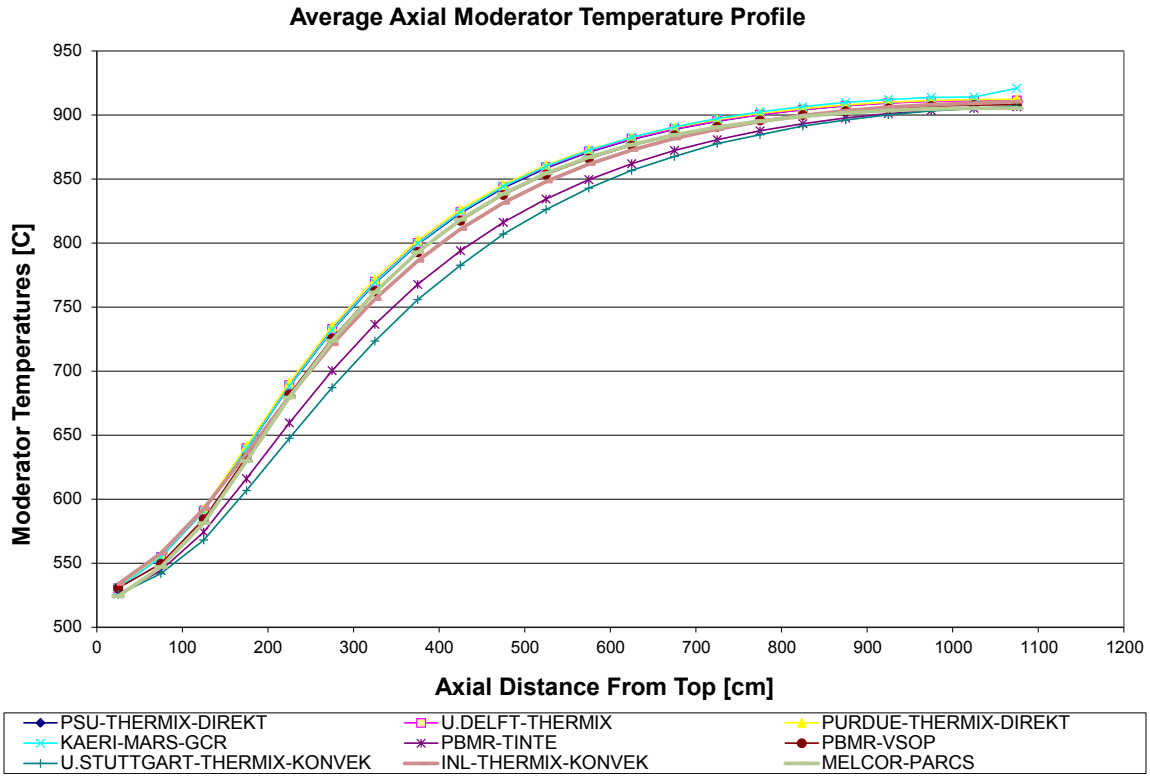

Figure 7.6: A comparison of computed profiles for average axial moderator temperature for the PBMR400 steady state benchmark. The values produced by this research are identified in the figure under the name 'MELCOR-PARCS'.

Here it can be seen that the average axial moderator temperature profile, produced from the "component" temperatures are also quite comparable to the moderator temperature profiles of the other codes that were tested in the benchmark. It shouldn't be too surprising that this moderator temperature profile agrees even better with the other codes than the for the fuel temperature profile, since graphite makes up a large percentage of the mass of the core control volumes.

Presented below is another graphical comparison of the coupled MELCOR-PARCS/AGREE to the other codes from the OECD/NEA PBMR400 benchmark. For this plot, the average radial temperature profiles for the fuel are compared.

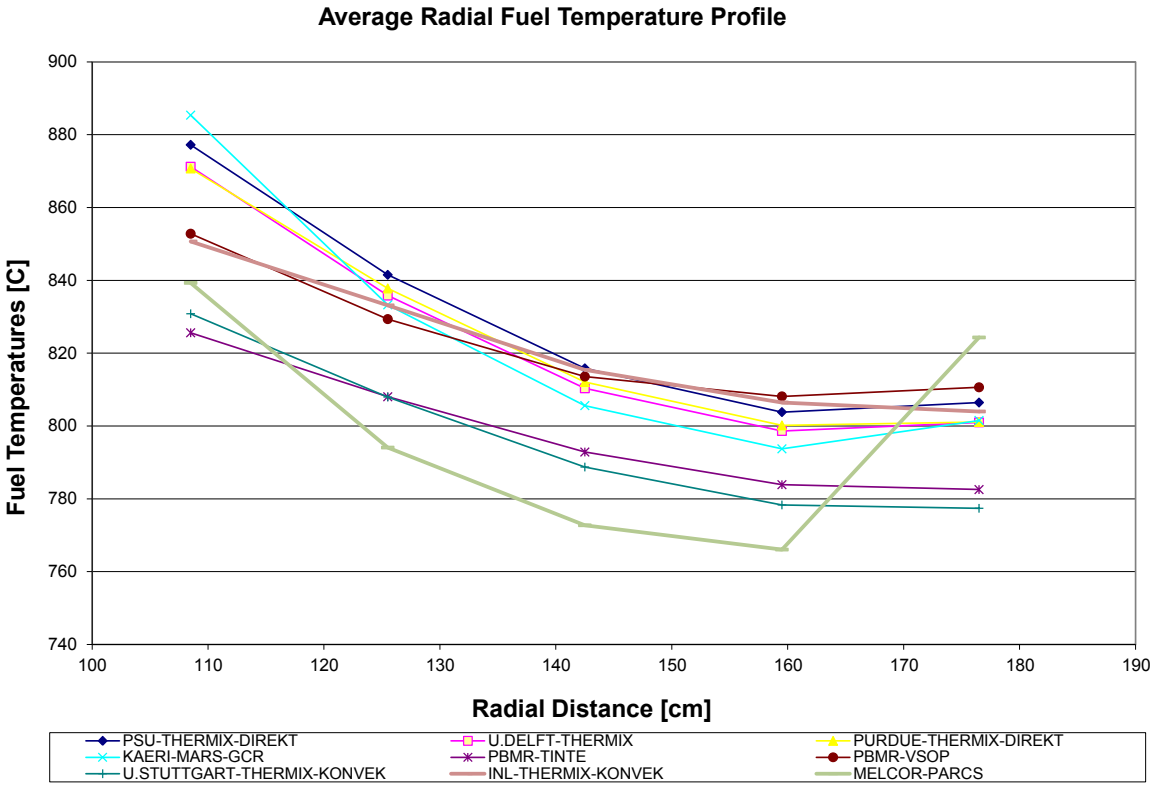

Figure 7.7: A comparison of computed profiles for average radial fuel temperature for the PBMR400 steady state benchmark. The values produced by this research are identified in the figure under the name 'MELCOR-PARCS'.

It seems for whatever reason, that the largest deviations between the output from the MELCOR-PARCS/AGREE coupling and the other codes, consistently appears upon examination of any radial profile. While the value at the innermost region of the fuel appears to be reasonable with respect to the other points, the other inner radial sections seem to consistently underestimate the value for the fuel temperature when compared with the other profiles. However, the most inconsistent feature by far, is the magnitude of the outermost radial temperature. While a number of the other participants also show a modest temperature gain in moving to the outermost section, this code coupling predicted a temperature increase at a much higher level.

The next plot is a comparison of the average radial temperature profiles for the

moderator. This is yet another one of the graphical comparisons of the coupled MELCOR-PARCS/AGREE to the other codes from the OECD/NEA PBMR400 benchmark.

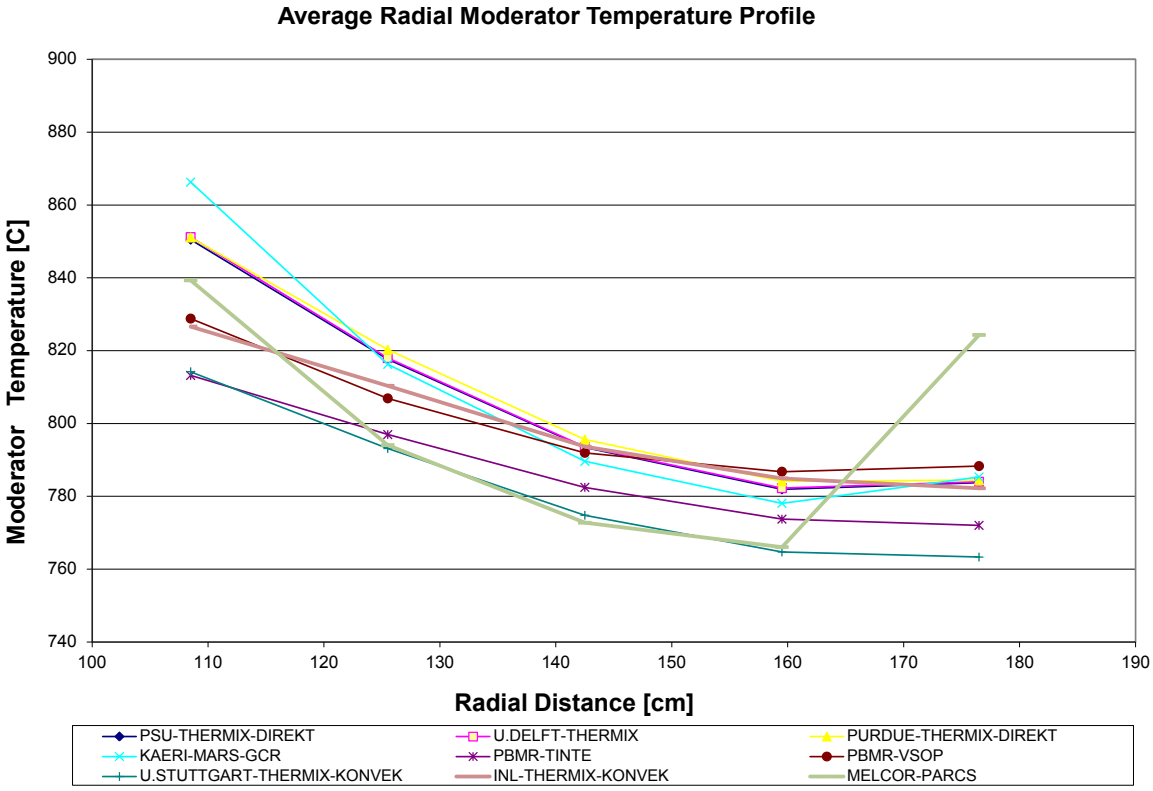

Figure 7.8: A comparison of computed profiles for average radial moderator temperature for the PBMR400 steady state benchmark. The values produced by this research are identified in the figure under the name 'MELCOR-PARCS'.

Since the same dataset was used between this plot and the previous one, it is no surprise that the trend of the data points is exactly the same. However, one noticeable difference between this plot and the previous, is how well, all but the radially outermost temperature value agree with the other participants. One aspect that is the same between both plots, is that the radially outermost data point seems to be quite a bit higher than the other values.

Similar to the OECD/NEA PBMR400 fuel benchmark data section (table 7.1),

table 7.2 is the OECD/NEA PBMR400 coolant benchmark data section, produced by the MELCOR-PARCS/AGREE coupling.

Helium / Coolant Temperatures (°C)						
MELCOR-PARCS	*108.5*	*125.5*	*142.5*	*159.5*	*176.5*	ave
25	510.51	506.62	505.62	504.82	507.82	507.078
75	526.48	518.01	515.28	513.11	518.19	518.214
125	550.04	535.55	530.59	526.8	536.5	535.896
175	581.28	560.01	552.84	548.28	570.16	562.514
225	618.24	590.23	580.89	576.08	610.58	595.204
275	657.86	623.85	612.29	607.34	652.4	630.748
325	697.48	658.52	644.76	639.65	693.2	666.722
375	735.15	692.35	676.5	671.19	731.23	701.284
425	769.65	724.05	706.28	700.74	765.48	733.24
475	800.37	752.81	733.36	727.56	795.52	761.924
525	827.12	778.29	757.39	751.32	821.33	787.09
575	850.02	800.43	778.29	771.97	843.12	808.766
625	869.37	819.39	796.2	789.65	861.28	827.178
675	885.54	835.44	811.38	804.62	876.23	842.642
725	898.94	848.91	824.14	817.19	888.41	855.518
775	909.95	860.14	834.77	827.66	898.24	866.152
825	918.93	869.42	843.57	836.33	906.05	874.86
875	926.17	877.05	850.8	843.45	912.15	881.924
925	931.94	883.25	856.68	849.25	916.8	887.584
975	936.43	888.21	861.39	853.91	920.19	892.026
1025	939.8	892.1	865.09	857.56	922.43	895.396
1075	942.36	895.65	868.63	860.78	920.09	897.502
ave	785.619545	745.921818	727.579091	721.784545	775.790909	

Table 7.2: The data section for an entry to the OECD/NEA PBMR400 steady state benchmark. This data is of average fluid temperatures as extracted from MELCOR, with the only modification being a change of units from Kelvin to Celcius. The values listed in yellow (horizontally and vertically) are average values for the radial rings and axial levels, respectively.

This set of data is used to calculate the average axial and radial values on the right and bottom of the table, respectively. The values used for this data set come from an average spatial "fluid" temperature distribution. This spatial temperature distribution like the fuel temperature distribution, is taken from the steady state

coupling output produced by MELCOR at a simulation time of 1000.0 seconds.

Presented below is another graphical comparison of the coupled MELCOR-PARCS/AGREE to the other codes from the OECD/NEA PBMR400 benchmark. For this plot, the average axial temperature profiles for the coolant are compared.

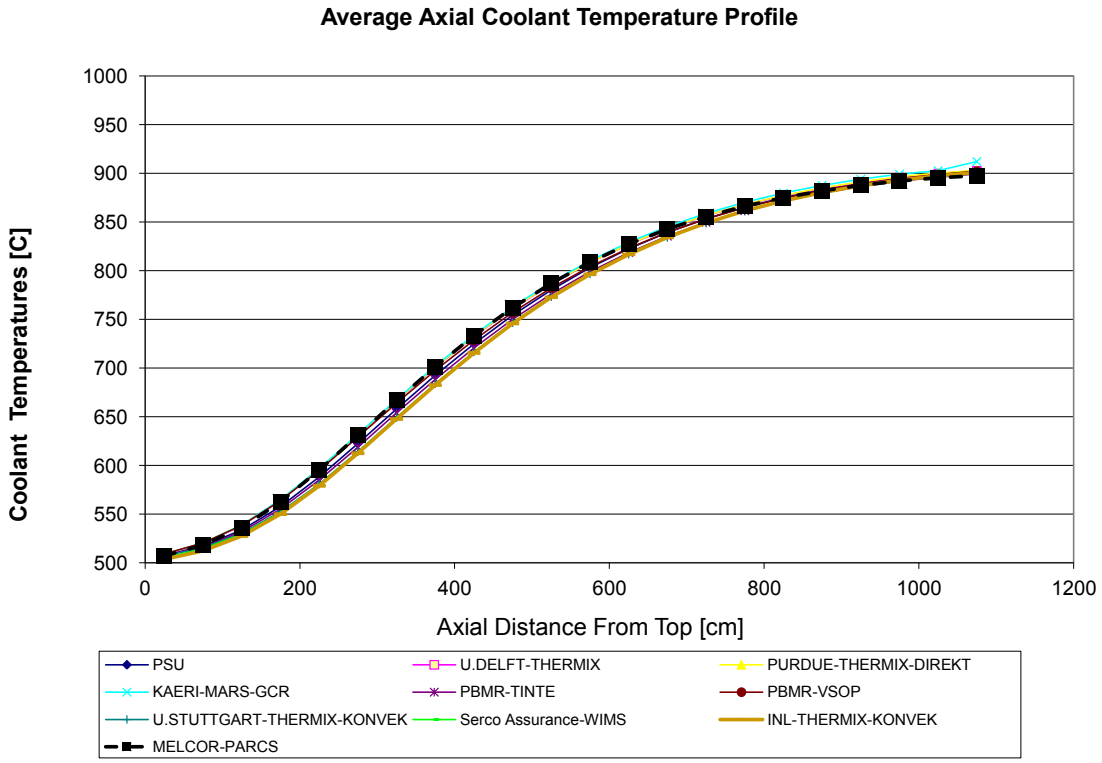

Figure 7.9: A comparison of computed profiles for average axial coolant temperature for the PBMR400 steady state benchmark. The values produced by this research are identified in the figure under the name 'MELCOR-PARCS'.

Here it can be seen that the average axial coolant temperature profile, produced from the "fluid" temperatures is very consistent to the coolant temperature profiles produced by the other codes that were tested in the benchmark.

The following plot is the last of the graphical comparisons of the coupled MELCOR-PARCS/AGREE to the other codes from the OECD/NEA PBMR400 benchmark.

In this plot, the average radial coolant temperature profiles of the various codes are compared.

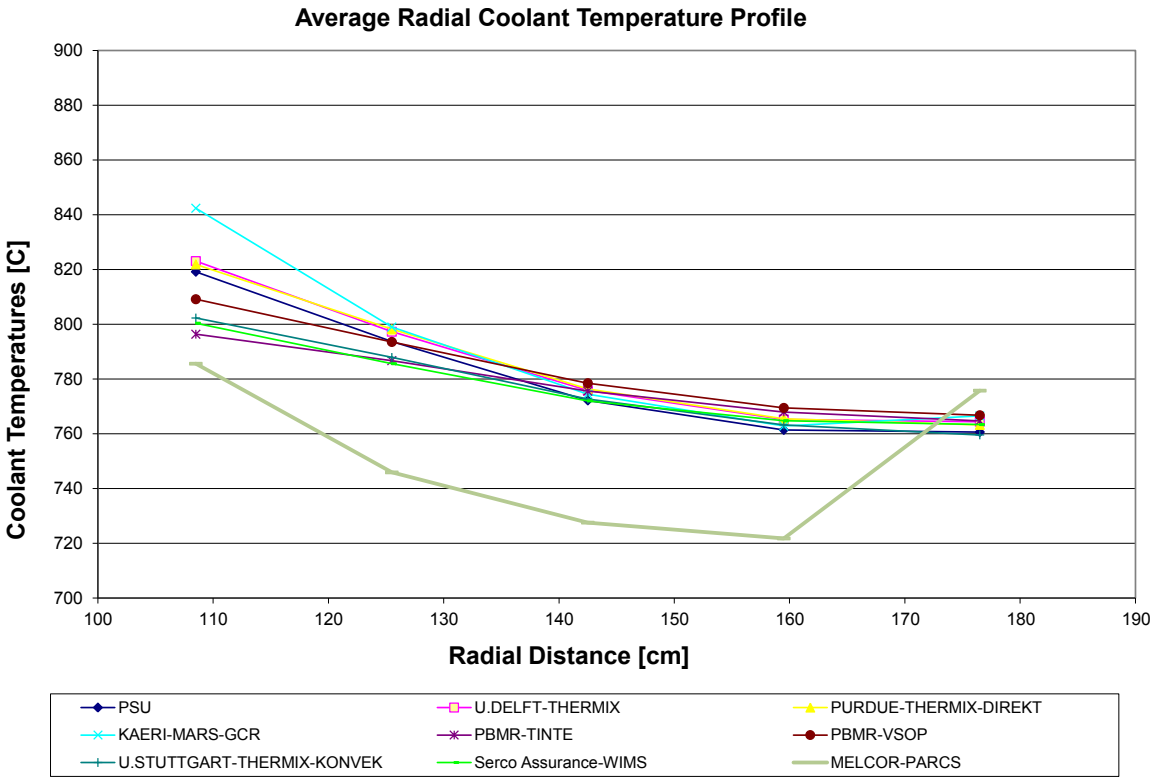

Figure 7.10: A comparison of computed profiles for average radial coolant temperature for the PBMR400 steady state benchmark. The values produced by this research are identified in the figure under the name 'MELCOR-PARCS'.

Here it can be seen that the average radial coolant temperature profile, produced from the "fluid" temperatures differs quite significantly from the profiles produced by the other codes that were tested in the benchmark. It is likely that whatever caused the average radial "component" temperature profile to have a similar shape, is also the reason behind the shape of this profile.

7.2 Coupled PBMR400 Total Control Rod Ejection Transient Simulation

The second example is of a coupled PBMR400 total control rod ejection transient simulation. In this demonstration, a PARCS/AGREE model first runs the

entire transient and then stops. After that simulation completes, the spatial power distribution from the first time step of the PARCS/AGREE run is extracted and used in the MELCOR input. Before this information can be used by MELCOR it must be converted to be in the style that MELCOR expects.

Next, the point kinetics information which was also extracted from the PARCS/AGREE output is then converted for use by MELCOR. By utilizing a MELCOR external data file (EDF) the value for the point kinetics model reactivity is specified to MELCOR as a function of time. After this, the main MELCOR input file is updated and other new data and input files created. Finally, this coupled example is completed by running MELCOR with the input deck in its current state.

The following section is in its entirety, the ovr script used to accomplish the described PBMR400 total control rod ejection transient coupling from PARCS/AGREE to MELCOR. The script is structured into a number of logical sections with comments enumerating and describing the sections Immediately following this, there is a more detailed explanation of each of these input sections.

7.2.1 Script for the PBMR400 Total Control Rod Ejection Transient Coupling

```
# ----- MINIMUM VERS. ovr 2.66
## 1 --- set executable paths first
parcs_path = "C:/mleimon/parcs/BIN/agree10.exe"
parcs.set_path(parcs_path)
mel_path = "C:/mleimon/melcor/melcor_executables/PC/windows/optimized"
melcor.set_path(mel_path)

## 2 --- now enter input file locations
parcs_inp = "input--parcs/inp/P400TR.inp"
mel_inp   = "input--melcor/pbmr400.inp"
```

```
## 3 --- execute parcs on input file
parcs.run(parcs_inp)

## 4 --- load output from parcs execution
output_parcs = parcs.load_output(parcs_inp)

## 5 --- describe the power mapping from parcs to melcor
pwr_map = [(0.0, (6,25), (10,4)), (2, 6, 400)]

## 6 --- use this maplist (which contains one map) to convert the power
mel_power = melcor.convert_power(output_parcs, [pwr_map])

#### GENERATE THE ‘COR_ZP,RP‘ POWER INPUT

## 7 --- create table for COR_ZP input
# these are the porosities....
por1 = 0.39
por2 = 0.20
# this name prefix below was just used to make the tabular input shorter
# alternatively, it could have been entered on all rows like the first
# entry in tab_ZP below.
n_pfx = "’COR-RAD-BND-A"

# similar to COR_ZP input  [[!Z, PORDP, IHSA], ... , [...]]
tab_ZP = [[-4.35, por1, "’COR-RAD-BND-A1’"],
          [ -4.0, por2, n_pfx + "2’"],
          [ -3.0, por2, n_pfx + "3’"],
          [ -2.0, por2, n_pfx + "4’"],
          [ -1.0, por2, n_pfx + "5’"],
          [  0.0, por1, n_pfx + "6’"],
          [  0.5, por1, n_pfx + "7’"],
```

```
          [  1.0, por1, n_pfx + "8'"],
          [  1.5, por1, n_pfx + "9'"],
          [  2.0, por1, n_pfx + "10'"],
          [  2.5, por1, n_pfx + "11'"],
          [  3.0, por1, n_pfx + "12'"],
          [  3.5, por1, n_pfx + "13'"],
          [  4.0, por1, n_pfx + "14'"],
          [  4.5, por1, n_pfx + "15'"],
          [  5.0, por1, n_pfx + "16'"],
          [  5.5, por1, n_pfx + "17'"],
          [  6.0, por1, n_pfx + "18'"],
          [  6.5, por1, n_pfx + "19'"],
          [  7.0, por1, n_pfx + "20'"],
          [  7.5, por1, n_pfx + "21'"],
          [  8.0, por1, n_pfx + "22'"],
          [  8.5, por1, n_pfx + "23'"],
          [  9.0, por1, n_pfx + "24'"],
          [  9.5, por1, n_pfx + "25'"],
          [ 10.0, por1, n_pfx + "26'"],
          [ 10.5, por1, n_pfx + "27'"],
          [ 11.0, por1, n_pfx + "28'"],
          [ 11.5, por2, n_pfx + "29'"]]

## 8 --- create table for COR_RP input
n_pfx = "'TOP-PLATE-R"

# like COR_RP input: [[!RINGR, IHSR, ICFCHN, ICFBYP], ..., [...]]
tab_RP = [[1.000, n_pfx + "1'", "NO",       "NO"],
          [1.170, n_pfx + "2'", "'FLDIR2'", "'FLDIR2'"],
          [1.340, n_pfx + "3'", "'FLDIR3'", "'FLDIR3'"],
          [1.510, n_pfx + "4'", "'FLDIR4'", "'FLDIR4'"],
```

```
          [1.680, n_pfx + "5'", "'FLDIR5'", "'FLDIR5'"],
          [1.850, n_pfx + "6'", "'FLDIR6'", "'FLDIR6'"],
          [2.436, n_pfx + "7'", "NO",       "NO"],
          [2.606, n_pfx + "8'", "NO",       "NO"]]

## 9 --- convert the internal melcor power datatype to a usable string input
pow_str = melcor.generate_input(mel_power, "COR_ZP,RP", tab_ZP, tab_RP,
                                mel_inp)

## 10 --- lets update the melcor input (with power info) at this point
m_inp = melcor.load_input(mel_inp)
m_inp = melcor.update(m_inp, pow_str, "tab", "replace")

## 11 --- at this point we need to get the point kinetics data from the parcs
## output and then generate input from it.
mel_pk = melcor.convert_pk(output_parcs, 0.0, 9000.0)

#####  POINT KINETICS SECTION
## 12 --- this is the COR_SC section, contains lambdas & betas
pk1_str = melcor.generate_input(mel_pk, "COR_SC")

## 13 --- this is the EDF-reactivity section, it produces the EDF data
pk2_str = melcor.generate_input(mel_pk, "EDF", "reactivity", 0.1)
EDF_loc = fs_dir_of(mel_inp)
EDF_loc = EDF_loc + "inp_rho_EDF.dat"
fs_write(EDF_loc, pk2_str, "overwrite")

## 14 --- this is the reactivity section, it produces EDF and CF code to use
## the EDF file produced in the preceding command.
react_loc = fs_dir_of(mel_inp)
react_loc = react_loc + "inp_react.inp"
```

```
pk3_str = melcor.generate_input(mel_pk, "reactivity", "DOLLARS", 999,
                                "inp_rho_EDF.dat")
fs_write(react_loc, pk3_str, "overwrite")
##### END OF POINT KINETICS SECTION

## 15 --- now we want to update the input with pk data
m_inp = melcor.update(m_inp, pk1_str, "tab")

## 16 --- now lets write out a new input file
m_inp_str = melcor.view(m_inp)
fs_write(mel_inp, m_inp_str, "overwrite")

## 17 --- now it is time to execute melcor
melcor.run(mel_inp)
```

7.2.2 Detailed Explanation of the Script

The following is a more detailed explanation of the various sections of the ovr script used for the transient coupling. These code sections will be referred to by their number, which can be seen at various points in the code (e.g. '`## 4 --- ...`', marks the start of the fourth code section).

The first two sections of this script define a number of paths and serve as the general setup for the problem. Section one of the script creates two strings: 'parcs_path' and 'mel_path' and then uses them as arguments to the *set_path* functions for both the PARCS and MELCOR modules respectively. This is done so that the interface understands the location of these executables. Section two of the script creates a couple of location strings which are then later used with run functions of the PARCS and MELCOR modules. While, it is not necessary that these variables be created before calling the run functions, it is done here to maximize readability and also to

allow easy reuse.

Together, the third and fourth sections run PARCS on the specified input file and subsequently load the output that is generated. The line denoting section three executes the version of PARCS previously specified by *parcs.set_path()* on the supplied argument, in this case 'parcs_inp'. In section four the command *parcs.load_output()* loads the output files corresponding to a corresponding PARCS input file, which is what is specified as the argument to this function. This function returns a *parcs__output* datatype and so the variable 'output_parcs' is assigned this value.

The fifth and sixth sections of the script both describe an equivalent core-to-core power mapping and then use this mapping to convert a PARCS spatial power distribution to one usable by MELCOR. The fifth section creates a maplist (a list of powermaps), which describes the power information mapping from a PARCS input to a corresponding MELCOR input. The first argument of the 'pwr_map' is (0.0, (6,25), (10,4)), this is information about the PARCS output. More specifically, the first value is the PARCS simulation time of the desired power information, the following two values describe two locations of the PARCS nodalization, a starting and ending point, respectively. The second value (2, 6, 400), is information about the MELCOR input. More specifically, the starting ring, starting level and starting control function number. In the sixth section *melcor.convert_power()* uses a parcs__output datatype in conjunction with a maplist to create a corresponding melcor__power datatype. This function returns a melcor__power datatype, thus the variable 'mel_power' is of this type.

The seventh and eighth sections of the script create a ZP_table and an RP_table, which are required to use the "COR_ZP,RP" input generation style of the melcor.generate_input function. The seventh section assigns a ZP_table as the value of variable 'tab_ZP'. As this information will be used to construct a MELCOR COR_ZP

input section (Axial Level Parameters), it resembles this to a degree. ZP_tables are lists of lists which then contain three arguments. These arguments are the following three MELCOR COR_ZP inputs: Z (the elevation of the lower axial level boundary), PORDP (porosity of debris in the axial level), and IHSA (the boundary heat structure name for the particular axial level). In the eighth section assigns an RP_table as the value of variable 'tab_RP'. As this information will be used to construct a MELCOR COR_RP input section (Radial Ring Parameters), it resembles this to a degree. RP_tables are lists of lists which then contain four arguments. These arguments are the following four MELCOR COR_RP inputs: RINGR (the outer radius of the ring), IHSR (name of the upper boundary heat strucutre for this ring), ICFCHN (name of the control function that the flow direction may be inferred from), and ICFBYP (name of the control function that the bypass flow direction may be inferred from).

The ninth code section generates an input string for the spatial power distribution for MELCOR. Here the function *melcor.generate_input()* is used by supplying it with a melcor__power datatype as well as a string "COR_ZP,RP", which specifies the style of input generation. Additionally, this input generation style requires three additional arguments: a ZP table, an RP table, and a location string for the melcor input file (the input file is used to grab the HSALT value from the HS_EOD for calculation of the dz values). This particular usage of *melcor.generate_input()* generates COR_ZP and COR_RP sections to describe the power distribution. This function returns a string representing a section of valid MELCOR input, this value is assigned to a new variable named 'pow_str'.

The tenth code section loads a MELCOR input file and then merges the spatial power input generated by the last code section into it. The first command here loads the MELCOR input file at location string 'mel_inp', the *melcor.load_input()* function returns a *melcor__input* datatype, which is then assigned to the variable

named ‘m_inp’. The second command here reassigns a new modified version of the previous created *melcor__input* datatype. The new version is created by use of the *melcor.update()* function, which in this case was used to replace existing tabular sections in the input file with newer versions that were included in the string ‘pow_str’. The arguments to this function are as follows: a *melcor__input* datatype, a string containing valid melcor input, a string describing the format of the valid melcor input (currenly only “tab” is supported), and lastly there is the “replace” string which changes the behavior of the update command so that it replaces existing tables with new tables.

The eleventh and twelfth sections of the script convert point kinetics information from PARCS output to a MELCOR module usable type and then using this converted information, generate MELCOR COR_SC input. In the eleventh section the *melcor.convert_pk()* function is used on a *parcs__output* datatype, stored within the variable named ‘output_parcs’. The other arguments supplied are: the PARCS simulation event time the and MELCOR simulation event time (in seconds). This function returns a *melcor__pk* datatype which is then assigned to a variable named ‘mel_pk’. In the twelfth section *melcor.generate_input()* is used with a *melcor__pk* datatype to produce a MELCOR COR_SC input section. This section contains sensitivity coefficient information such as the lambda and beta values for the point kinetics equations. This function returns a string containing valid MELCOR input which is then assigned to the variable named ‘pk1_str’.

The 13th code section is used to generate a MELCOR external data file (EDF) filled with reactivity information. The first command uses *melcor.generate_input()* on a *melcor__pk* datatype with the following three arguments: a string containing input style (in this case “EDF”), a string specifying which time dependent variable to use (in this case “reactivity” is used), and the last option is an optional argument

specifying a minimum time interval between data points (in seconds). This function returns a string containing valid EDF input which is then assigned to the variable named 'pk2_str'. The second command uses *fs_dir_of()* on the MELCOR input file location string, to get the directory in which it is contained. This command returns a location string of that directory to the variable 'EDF_loc'. The following command appends the desired name of the EDF to the directory location, the end result of this will be that the EDF will be created in the same directory as the MELCOR input file it is being used with. The final command uses the *fs_write()* function to write the contents of the string in variable 'pk2_str' to a file at location 'power_file'. The last argument, "overwrite", is an optional argument which specifies the unprompted overwriting of preexisting files at the specified location.

The 14th code section is used to generate a MELCOR input file which connects the reactivity information contained in the EDF to a usable CF. The first command here uses *fs_dir_of()* on the MELCOR input file location string, to get the directory in which it is contained. This command returns a location string of that directory to the variable 'react_loc'. The following command appends the desired name of the output file to the directory location, the end result of this will be that the new input file will be created in the same directory as the MELCOR input file it is being used with. Next *melcor.generate_input()* is used on a *melcor__pk* datatype stored within the variable named 'mel_pk'. In this case the following arguments are supplied: a string containing input style (in this case "reactivity"), a string containing the name of the CF which supplies the rho value (here it is "DOLLARS"), next is an integer that denotes which control function number to start with (values will be enumerated up from this), the last argument is a string containing the name of the input file. The final command uses the *fs_write()* function to write the contents of the string in variable 'pk3_str' to a file at location 'react_loc'. The last argument, "overwrite",

is an optional argument which specifies the unprompted overwriting of preexisting files at the specified location.

The 15th and 16th sections of the script once again updates the *melcor__input* datatype and then writes the resultant input to a file. In the 15th section, this command reassigns a newly modified version of the *melcor__input* datatype stored in the variable named 'm_inp'. The new version is created by use of the *melcor.update()* function, which in this case was used to merge contents of existing tabular sections in the input file with newer versions that were included in the string 'pk1_str'. The arguments to this function are as follows: a *melcor__input* datatype, a string containing valid MELCOR input, and a string describing the format of the valid MELCOR input (in this case "tab"). The 16th section writes an updated version of the MELCOR input file which includes all of the changes made by the *melcor.update()* function. First, *melcor.view()* function is used on the *melcor__input* datatype stored in the variable named 'm_inp', this function returns a string representing the contents of the input file, which is then stored in the variable named "m_inp_str". The last command uses the *fs_write()* function to write the contents of the string in variable 'm_inp_str' to a file at location 'mel_inp'. The last argument, "overwrite", is an optional argument which specifies the unprompted overwriting of preexisting files at the specified location.

Finally, the last section includes a single function which executes the MELCOR executable. More specifically, it first runs MELGEN executable on the supplied input file and then subsequently, the MELCOR executable. This effectively completes the transient coupling procedure.

7.2.3 Results and Analysis

Presented here are a number of plots generated from the output of the coupled total control rod ejection transient simulation. As was the case with the coupled steady state simulation, the same spatial power information from PARCS was coupled into MELCOR for this exercise. In addition to the spatial power information, point kinetics information was also coupled in this example, this includes the resultant reactivity spike due to the control rod ejection as calculated by PARCS/AGREE. As such, the focus of this example will be more on the transient aspects of the results. Another reason for there being less emphasis on spatial temperature analysis with the transient example is because the nodal fission power production fractions must be set by MELGEN to operate correctly with the point kinetics model of MELCOR. When set by MELGEN, these particular fractional values can not be modified again by MELCOR and so, the nodal fission power production fractions must stay the same throughout the course of the entire transient.

For the transient exercise, the MELCOR model operates in a steady-state manner for 9000 seconds of simulation, to allow the solution more than enough time to stabilize. It is at this point in the MELCOR simulation that the control rod ejection transient occurs. The simulation model for PARCS/AGREE is different in this respect, as the transient event begins immediately at a simulation time of 0.0 seconds. The resulting reactivity values from PARCS/AGREE output for this model were translated to occur 9000 seconds later within the simulation time of the corresponding MELCOR model.

The figure presented below is a plot of the transient reactivity, as generated from the PARCS/AGREE output. According to the PARCS/AGREE output, this system reaches a maximum value of reactivity of 3.121$ at the transient time of 0.147 seconds.

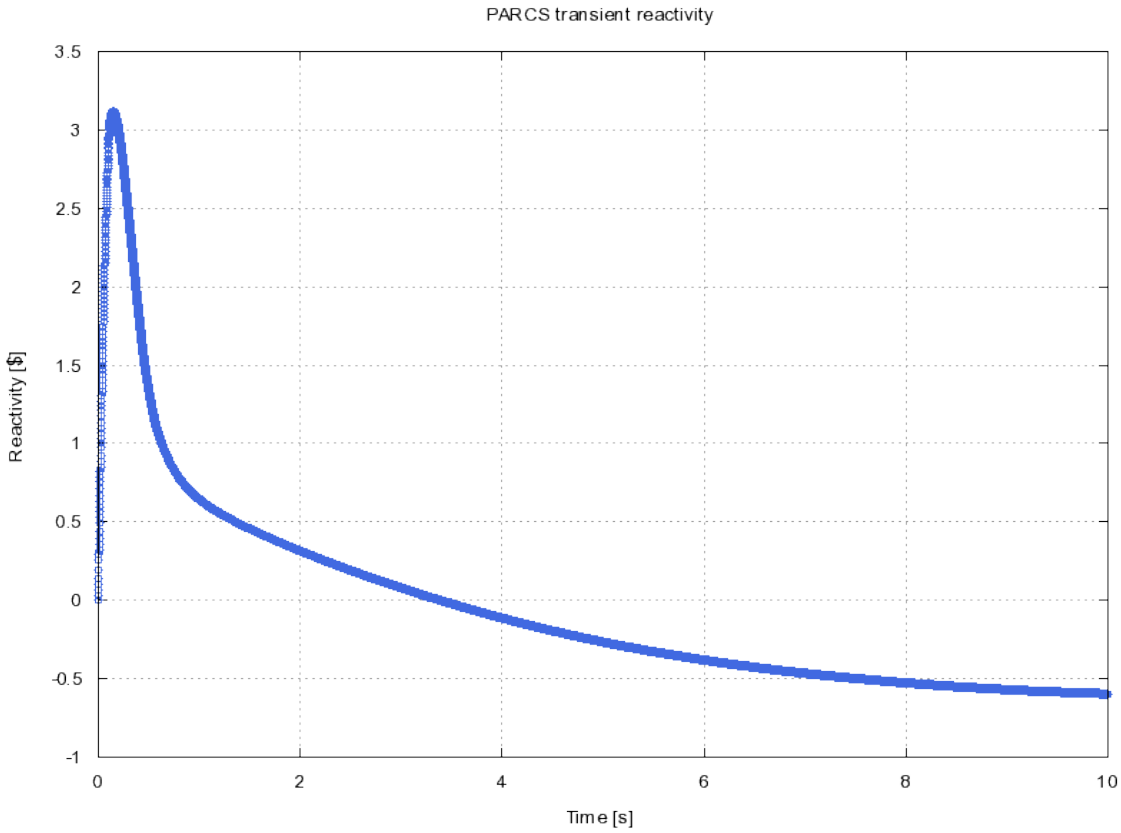

Figure 7.11: A plot of the reactivity duing a total control rod ejection transient. This time listed here is the PARCS simulation time.

Notice that beyond the transient time of 3.387 seconds, the value for reactivity is a negative value. Once coupled into MELCOR, this transient event will occur at the MELCOR simulation time of 9000 seconds.

Below is a figure depicting the total power production within the reactor core of the coupled MELCOR model simulation.

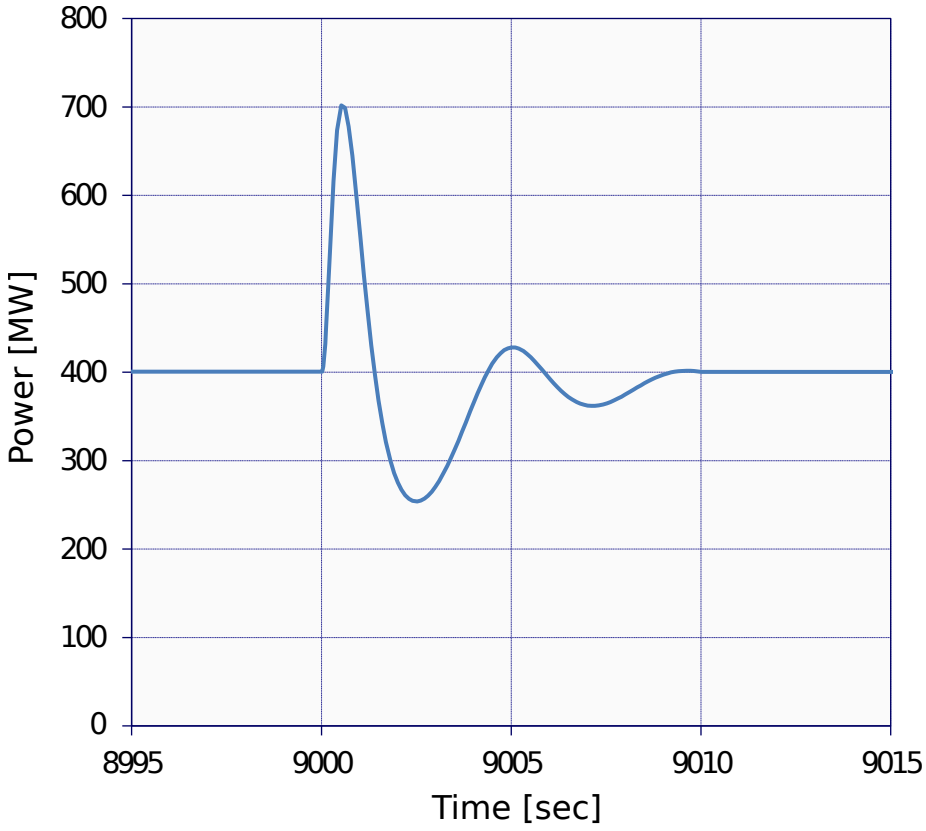

Figure 7.12: A plot of the total reactor power for the total control rod ejection transient simulation. This time listed here is the MELCOR simulation time. It can be seen that the transient event occurs in the MELCOR simulation at t = 9000s.

Here it can be seen that the maximum total power output for the core is just over 700 MW, which the simulation reaches for only a split second. Following the initial power spike, the power output oscillates and dampens for a number of seconds until returning once again to a stable level (this is likely because the reactivity information from PARCS/AGREE was only supplied for that 10 second interval). In any case, the values of most interest and the values that are further investigated here occur within this 10 second span.

The plot included in the figure below shows the maximum recorded value for MELCOR component temperature as a function of time. While the resultant shape of this plot follows what would be expected from a power excursion like the one shown in the previous figure, there are a number of issues with the values and the magnitudes recorded here.

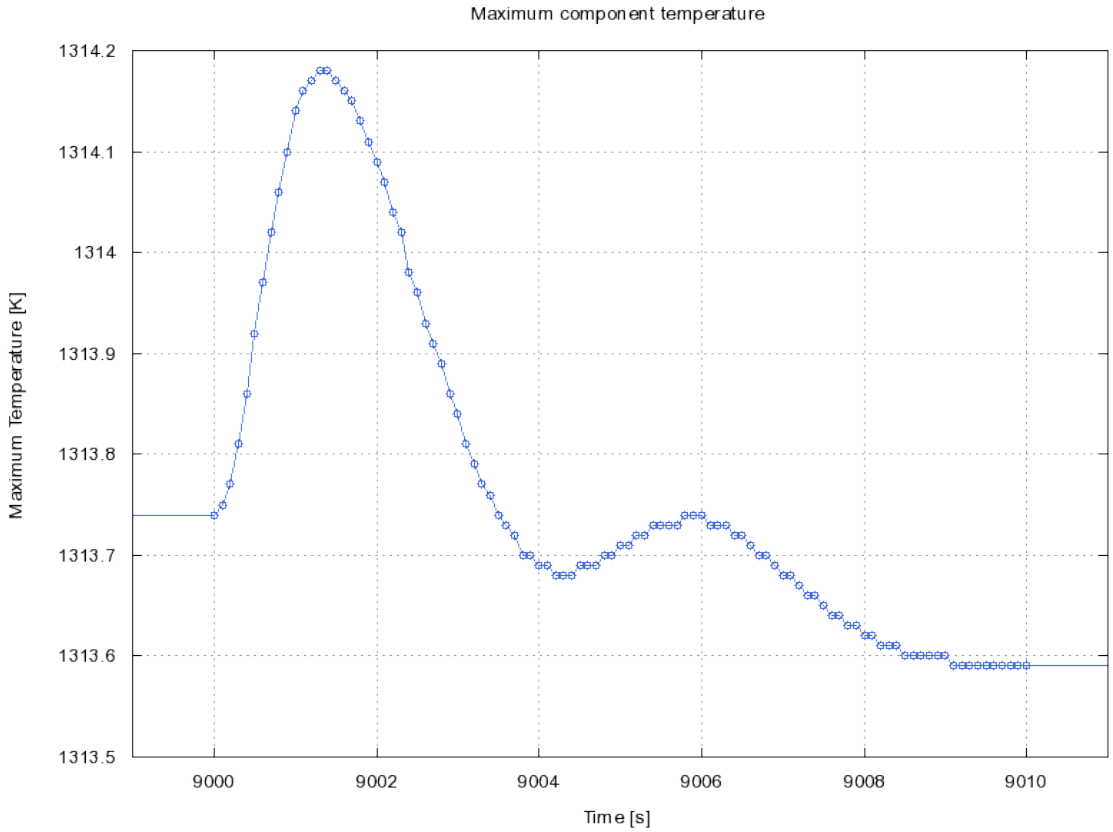

Figure 7.13: A plot of the maximum component temperature during the interval around the total control rod ejection transient simulation.

The first issue with this plot is regarding the initial temperature prior to the transient. This plot shows that the initial temperature prior to the transient is approximately 1313.75K, this is quite higher than the stabilized maximum temperature value from the steady state case, which was approximately 1223K. One likely cause of this discrepancy stems from the inherently different methods in which these

two models specify the coupled spatial power production to MELCOR. In any case, these two methods of specification should be equivalent. The second issue with these results are with the magnitude of the resultant temperature spike. It seems quite inconceivable that a 3+$ reactivity transient would only result in only approximately a 0.4K increase in the maximum fuel temperature. However, one thing to keep in mind is that this plot is showing the maximum average "component" temperature, which doesn't necessarily represent the fuel temperature appropriately here.

The next figure shown below, is a plot of the core average spatial component temperatures which was extracted from the timestep containing the maximum component temperature value from the entire simulation.

The apparent differences between this distribution and that of the stabilized steady-state model is also cause for concern. Since the temperature profiles of the steady-state model seemed to agree quite well with the other participants, it could be an indicator that some aspect of the method for coupling spatial power information for transient events is even more inconsistent.

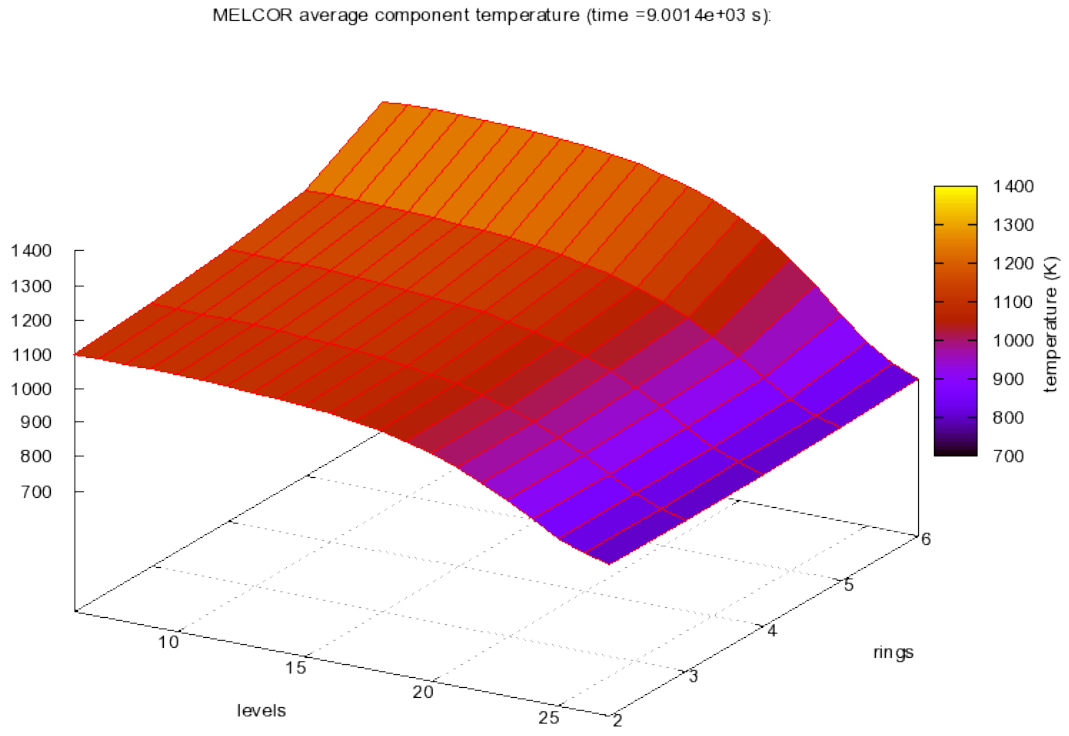

Figure 7.14: A plot of the average spatial component temperatures for a total control rod ejection transient simulation. This distribution came from a MELCOR output file from a simulation time of 9001.4 s. This plot is taken from one of the times with the highest value for maximum component temperature.

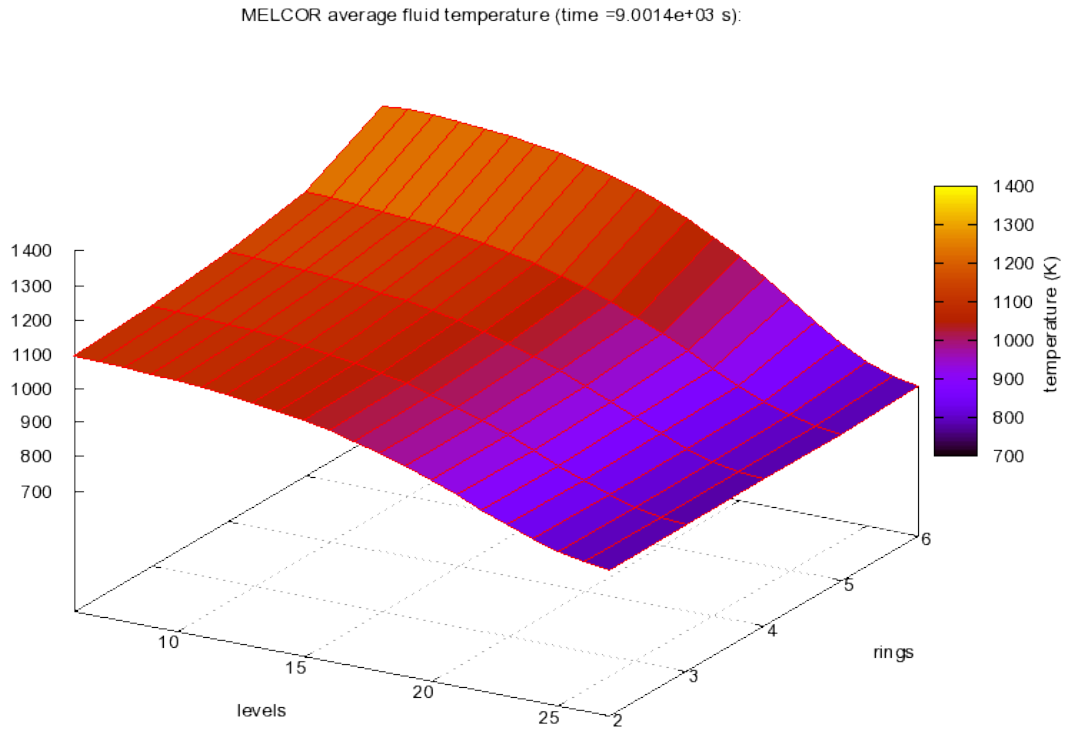

Figure 7.15: A plot of the average spatial fluid temperatures for a total control rod ejection transient simulation. This distribution came from a MELCOR output file from a simulation time of 9001.4 s. This plot is taken from one of the times with the highest value for maximum component temperature.

The figure shown above, is a plot of the core average spatial fluid temperatures which was extracted from the timestep containing the maximum component temperature value from the entire simulation. Similar to the previous figure, this plot suggests that the highest temperatures occur within the sixth radial ring of the MELCOR model, which is the outermost radial ring of the core. This seems inconsistent with what one would expect the distribution to look like at the height of a reactivity insertion transient.

8. CONCLUSIONS AND RECOMMENDATIONS

8.1 Conclusions

As a result of this research, a new code was written to couple the reactor physics code PARCS-AGREE to the systems level code MELCOR. This new code is a one directional coupling from PARCS-AGREE to MELCOR. The information coupled between the two codes includes: spatial power information, point kinetics parameters, as well as reactivity values in the case of a transient. To demonstrate the new coupling code, a couple of tests were devised and run. These tests were based on the OECD/NEA PBMR400 benchmark. More specifically, a coupled steady state case was run as well as a coupled total control rod ejection transient. The original contribution made to the field of nuclear engineering by this research is in the development of the first coupling code for PARCS-AGREE to MELCOR for HTGR applications.

8.1.1 Accomplishments of This Research

The coupling interface was written using version 3 of the Python language. This choice was made so that there would be long term support of the language of the code as well as to enable cross platform operation. While the interface currently runs on Linux, OSX and Windows, only the Windows version of the coupling is currently feature complete.

Operationally the design of the code is in a set of modules, each provided a set of micro-routines. Instead of hard coding the coupling procedures into the interface, the interface provides instead includes a set micro-routines simplify the translation and passing of information between the supported codes. This design feature gives the code user more flexibility to adapt to the inputs of the coupled codes as opposed to forcing the inputs of the coupled codes to adapt significantly to utilize the coupling

interface. This additionally allows the user to better utilize the post-processing capabilities of the code.

Additional features of the coupling code include an advanced error reporting system as well as some functionality for output post-processing. The included error reporting routines are designed to generate multiple reverse-ordered messages at varying operational levels to provide the user with enough information to quickly solve a problem. Additionally, errors during an interactive session are recoverable and error messages produced while interpreting a file include additional information about the line on which the error occurred.

The included output post-processing capabilities allow the user to quickly extract and or visualize the output of coupled simulation runs, or to even analyze the output of uncoupled simulation code output. Because the coupling code utilizes unstructured input, this allows the user to either: combine some amount of post processing directly into coupling routines, or utilize the coupling code solely as an output analysis tool. Currently, the plotting of output is accomplished by leveraging a local installation of the graphing utility, gnuplot. One of the other included functions allows for the production of a human readable output and data tables which are easily importable into spreadsheet programs.

To test the implementation of the coupling interface, a couple of examples were selected for testing. The first of the demonstrations tested a steady-state spatial power coupling from PARCS/AGREE to MELCOR. Because of the large base of preexisting knowledge and availability of prior benchmark efforts, both of the selected test cases examine a coupling between equivalent PBMR400 models. For this steady-state coupling demonstration, the focus of the analysis was on the stabilization of the MELCOR model as well as, an examination of the various spatial temperature distributions. A significant portion of the analysis of this demonstration is reserved

for comparisons with the OECD/NEA PBMR400 benchmark data.

The second demonstration used to test the coupling interface was a coupled total control rod ejection transient. Like the steady-state spatial power coupling, this transient was a coupling from PARCS/AGREE to MELCOR. In addition to utilizing a spatial power coupling as with the steady-state case, this demonstration also included coupled point kinetics information as well as the resultant transient reactivity values. The transient reactivity values calculated by PARCS/AGREE were translated to an equivalent simulation time for the MELCOR model which allowed it sufficient time to stabilize. For the total control rod ejection transient coupling demonstration, the focus of the analysis was mainly on the transient aspects of the resultant coupled simulation.

8.1.2 Main Findings

A number of interesting points can be drawn from the analysis of the results of the two coupling demonstrations. One of the aspects of the steady-state coupling that was examined was the initial stabilization of the MELCOR model for the PBMR400. Due to the design of MELCOR, these simulation models must be run initially for some quantity of time to allow for the solution to stabilize. Following the stabilization of the coupled MELCOR steady-state model, a couple of average spatial temperature distributions were examined. As would be expected, the nodal volume located at the radially innermost core section furthest from the coolant inlet into the core reached the hottest temperatures. Both of the average "component" and "fluid" temperature distributions produced by MELCOR did indeed have similar profiles overall. The major difference between the two profiles was that the magnitude of the "fluid" temperature profile lagged behind that of the "component" temperature profile for any given nodal volume in the direction of the coolant flow. However, this is exactly

the relationship that would be expected between a coolant that is removing heat from some other component.

The values used for the MELCOR average control volume temperatures were extracted from 'COR' sections of the plaintext program output. In these sections, these spatial temperature sections referred to the two constituent items within each control volume as the "fluid" and "component". While the descriptions of these two media are vague at best, testing and comparison with the OECD/NEA PBMR400 benchmarks showed that "component" medium is likely a mixture of both the fuel and moderator, though comprised more of moderator than of fuel. Similarly, the "fluid" media compared reasonably well with the coolant values produced by the other participants of the OECD/NEA PBMR400 benchmark.

While it may be possible that less homogenized sets of spatial temperature information are stored within MELCOR output plot files, the usage of such information was not practical for this research. The information of the MELCOR output plot files is stored in a proprietary and closed source format. Since the specification required to extract information directly from a MELCOR plot file was not shared with this research effort, it was impossible to implement functionality to directly read these files. As such, the only publicly available methods to access the information stored within these files were all unscripable. Since one of the original goals of this design was to eventually support bi-directional couplings between PARCS/AGREE and MELCOR, the reliance upon any unscriptable step would effectively make the entire process unscriptable as a whole. It is because of this reason, that the potential usage of plot file information was removed from consideration.

Looking more specifically at the OECD/NEA PBMR400 benchmark comparisons, there were a number of other interesting points. Since only a single set of average spatial "component" temperatures were available, this single set of data was

used for both the comparisons with the fuel and moderator temperatures within the benchmark results. In both cases, the average axial temperature profiles agreed quite with both the fuel and moderator profiles of the other participants. On the other hand, the average radial temperature profiles diverged fairly significantly in comparison with both the fuel and moderator profiles of the other participants. The most seemingly irregular data point of these radial profile comparisons being the radially outermost datapoint, where the results for the PARCS/AGREE-MELCOR coupling consistently over-predicted the value with respect to the other participants.

Values for the average spatial coolant temperatures were also used as a part of the steady-state coupling comparison with the OECD/NEA PBMR400 benchmark. Like with the moderator and fuel comparisons, the average axial coolant temperature profile agreed quite well with the profiles produced by the other participants. Similarly, the average radial coolant temperature profile diverged quite significantly in comparison with the profiles produced by the other participants. It was actually with this average radial coolant temperature profile comparison, that the PARCS/AGREE-MELCOR coupling values differed the most from the other participants.

For the case of the coupled total control rod ejection transient, a different sort of points were analyzed. To calculate the neutronic behavior of the reactor during the transient event, PARCS/AGREE was first used to simulate the event. The PARCS/AGREE simulation suggests that the system reactivity reaches a peak value of 3.122$ at 0.147 seconds into the transient. After 3.387 seconds into the transient, the value for reactivity goes and stays negative. When coupled into MELCOR, this reactivity insertion resulted in a peak value for total core power production of approximately 700 MW. During the course of the coupled transient event, the total core power production spikes and then oscillates in a heavily dampened manner, approximately returning to the initial power level by the end of the 10 second transient.

The next focus of the transient demonstration analysis was on temperature values. The general shape of the plot showing the maximum recorded MELCOR component temperature as a function of time, did reflect the sort of reaction that would be expected by a power excursion of the type depicted in figure 7.12. However, looking beyond the general shape that would be expected of this function, there were a number of problems regarding the values and magnitude of the resultant temperature spike. The value of the initial maximum component temperature immediately prior to the transient differs significantly from the maximum temperature value obtained from the steady-state model after stabilization. These differing temperature values are 1313.75K and 1223K, for the transient model and the steady-state model, respectively. While there are a number of possibilities which could explain this difference, one likely posibility is that there is some inconsistency between the spatial power coupling method used for the steady-state coupling versus the method used by the transient method. As a reminder, these two methods are different, because the point kinetics model used by MELCOR does not behave correctly if the steady-state power specification method is used.

The last portion of the transient demonstration to be analyzed was the MELCOR average spatial "component" and "fluid" temperature distributions that were produced from the timestep containing the highest maximum component temperature value. While the values of these spatial temperature plots were consistent with the values of the transient maximum component temperature plot, they were quite inconsistent with the equivalent plots generated by the steady-state model. Not only are the overall magnitudes of this plot far in excess of that for the steady-state model, however the larger concern is actually regarding the shape of the overall profile. Given that the limitations of the point kinetics model for MELCOR require that the nodal fission power production fractions remain fixed, discrepancies with any single

temperature plot could hint at there being disproportionate power production for the entire duration of the simulation.

8.2 Recommendations for Future Research

Over the course of developing, using, and analyzing the resultant output from the developed coupling code, a number of points for future research have been accumulated. These future points of research fall into one of several categories: resolution of the peculiarities demonstrated in the output from coupled simulation runs, acquisition of less general temperature information from MELCOR, and the development and comparison of models utilizing different core nodalizations. The single goal behind all of the following recommendations is to improve the quality of the results produced by the coupling efforts.

The first set of recommendations for future research are focused on simply addressing a number of the peculiarities demonstrated in the output from coupled simulation runs. Out of all of the temperature profile comparisons between the developed coupling code and the other participants of the OECD/NEA PBMR400 benchmark, it was the radial profile comparisons with which the largest deviations occurred between MELCOR-PARCS/AGREE and the other participants. Currently, there is a problematic difference between the maximum component temperature value obtained from the resultant MELCOR steady-state model after being allowed to stabilize and the maximum component temperature value just prior to the start of the resultant MELCOR transient model. A task potentially linked to the previous issue is to resolve the apparent spatial temperature profile differences between the steady-state implementation and the transient implementation. The investigation and subsequent addressing these issues would certainly improve the viability of such a coupling.

As was mentioned previously, due to a number of unresolvable constraints a

potentially suboptimal data source was used to extract spatial temperature information from MELCOR. While this choice allowed the code to be developed and operate within design specifications, the data that is used appears to be overly general insomuch as it may not be usable for determination of important issues such as incipient fuel melt. Possibly the best option towards addressing this issue may be, another attempt by another party to obtain the MELCOR plotfile specifications, thus allowing for the development of routines to directly access this information within an external coupling program. While potentially more of a hack than the previous solution, the investigation and subsequent implementation of a clever set of MELCOR control functions could potentially be used to write various bits of data to numerous external output files, thereby accomplishing the same goal.

In the plot of the maximum average "component" temperature versus time for the transient coupling, it shows an unbelievably minor temperature spike in the midst of a 3+$ reactivity insertion. It is quite possible that this value appears so low because, the effects of any possible temperature spikes are lost in an overly course core nodalization. Obviously for the demonstrations done in this thesis, the nodalization was chosen to be the same as that used in the OECD/NEA PBMR400 benchmark so that comparisons could be made. However, this choice in conjunction with the joint treatment of fuel and graphite in a single component temperature output may have resulted in an effective loss of pertinent results. It may be possible to improve the quality of the results simply by increasing the number of nodes within the core region.

REFERENCES

[1] High-Temperature Gas-Cooled Reactor (HTGR) NRC Research Plan. Technical Report (ML110310182), NRC. pbadupws.nrc.gov/docs/ML1103/ML110310182.pdf.

[2] Charter of the Generation IV International Forum. pages 7–16, 2001. gen-4.org/PDFs/GIFcharter.pdf.

[3] Energy Policy Act of 2005. Pub. L. No 109-58. gpo.gov/fdsys/pkg/PLAW-109publ58/html/PLAW-109publ58.htm, July 2005.

[4] Development of Design and Simulation Model and Safety Study of Large-Scale Hydrogen Production Using Nuclear Power. Technical Report (SAND-2007-6218), Sandia National Laboratories, October 2007. page 199. melcor.sandia.gov/techreports/076218.pdf.

[5] J. Bouchard. Generation IV Advanced Nuclear Energy Systems. *Nuclear Plant Journal*, 26(5):3, September-October 2008.

[6] TRACE code development and assessment project. *TRACE V5.0 THEORY MANUAL - Equations, Solution Methods, and Physical Models.* UNITED STATES NUCLEAR REGULATORY COMISSION. DRAFT. pbadupws.nrc.gov/docs/ML0710/ML071000097.pdf.

[7] Nuclear Regulatory Commission. Advisory Committee on Reactor Safeguards Future Plant Design Subcommittee. Official Transcript of Proceedings, April 5, 2011. pages 106-108. pbadupws.nrc.gov/docs/ML1111/ML111110657.pdf.

[8] S. Dadzie and J. Meolans. Anisotrpic Scattering Kernel : Generalized and Modified Maxwell Boundary Conditions. *Journal of Mathematical Physics*, 45(5):1804–1819, 2004.

[9] M. DeHart and S. Bowman. Improved Radiochemical Assay Analyses Using TRITON Depletion Sequences in SCALE. In *IAEA Conference Paper*. Oak Ridge National Laboratory, March 2006. `www.ornl.gov/~webworks/cppr/y2001/pres/124734.pdf`.

[10] T. Downar, Y. Xu, and V. Seker. *PARCS v3.0 U.S. NRC Core Neutronics Simulator THEORY MANUAL*. Department of Nuclear Engineering and Radiological Sciences - University of Michigan, December 2009. DRAFT. pages 109-110 `pbadupws.nrc.gov/docs/ML1016/ML101610117.pdf`.

[11] T. Downar, Y. Xu, and V. Seker. *PARCS v3.0 U.S. NRC Core Neutronics Simulator USER MANUAL*. Department of Nuclear Engineering and Radiological Sciences — University of Michigan, Ann Arbor, MI, December 2009. DRAFT. `pbadupws.nrc.gov/docs/ML1016/ML101610098.pdf`.

[12] C. Fletcher and R. Shultz. *RELAP5/MOD3 CODE MANUAL - Volume V: User's Guidelines*. Idaho National Engineering Laboratory, Lockheed Idaho Technologies Company, Idaho Falls, Idaho 83415, relap5/mod3.2 edition, June 1995. `edasolutions.com/old/RELAP5/manuals/rv5.pdf`.

[13] R. Gauntt, R. Cole, C. Erickson, R. Gido, R. Gasser, S. Rodriguez, and M. Young. *MELCOR Computer Code Manuals*. Sandia National Laboratories, Albuquerque, NM 87185-0739, May 2001. Version 1.8.5. page iii `melcor.sandia.gov/techreports/010929p.pdf`.

[14] S. Goluoglu and M. Williams, editors. *Modeling Doubly Heterogeneous Systems in SCALE*, American Nuclear Society Winter Meeting. Oak Ridge National Laboratory, November 2005. www.ornl.gov/~webworks/cppr/y2005/pres/123639.pdf.

[15] Y. Hassan. Large Eddy Simulation in Pebble Bed Gas Cooled Core Reactors. OECD-NEA Reports. OECD-NEA, September 2007. www.sciencedirect.com/science/article/pii/S0029549307003378.

[16] D. Ingham and I. Pop, editors. *Transport Phenomena In Porous Media*, volume III. Elsevier: The Boulevard, Langford Lane Kidlington, Oxford OX5 1GB, UK, 2005. pages 147-148.

[17] A. Khaled and K. Vafai. The Role of Porous Media in Modeling Flow and Heat Transfer in Biological Tissues. *International Journal of Heat and Mass Transfer*, 43:4989–5003, 2003.

[18] J. Lieberoth and A. Stojadinovic. Neutron Streaming in Pebble Beds. *Nuclear Science and Engineering*, 76:336–344, 1980.

[19] United States. Dept. of Energy and United States. Nuclear Regulatory Commission. *Next Generation Nuclear Plant Licensing Strategy: A Report to Congress*. U.S. Department of Energy, 2008.

[20] J. Rhodes, K. Smith, and D. Lee. CASMO-5 Development and Applications. In *PHYSOR-2006, ANS Topical Meeting on Reactor Physics*. Studsvik Scandpower Inc. www.studsvikscandpower.com/documents/publications/c5.physor2006.pdf.

[21] S. Rodriguez, R. Gauntt, and R. Cole. Development of Design and Simulation Model and Safety Study of Large-Scale Hydrogen Production Using Nuclear Power. Technical report, Sandia National Laboratories, October 2007. pages 199-200. melcor.sandia.gov/techreports/076218.pdf.

[22] Sandia National Laboratories. *MELCOR Computer Code Manuals. Vol. 2 Reference Manual*, October 2009. Working Version: Revision 1237. pages CORRM-174 - 175.

[23] R. Scarlat. Pebble Bed Heat Transfer Particle-to-Fluid Heat Convection. (Presentation) Thermal Hydraulics Laboratory, Department of Nuclear Engineering, Berkeley pb-ahtr.nuc.berkeley.edu/PebbleBedHeatTransferGroupMeetingPresentation.pdf, February 2009.

[24] V. Seker. *Multiphysics Mehtods Development for High Temperature Gas Reactor Analysis*. PhD thesis, Purdue University, November 2007.

[25] T. Simeonov and C. Wemple. HELIOS-2: Benchmarking Against Hexagonal Lattices. In *18th AER Symposium on VVER Reactor Physics and Reactor Safety*. Studsvik Scandpower, 2008. www.studsvikscandpower.com/documents/publications/helios2-tic.pdf.

[26] W. Stacey. *Nuclear Reactor Physics*. Wiley-VCH: 111 River Street Hoboken, NJ 07030-5774, 2nd edition, 2007. page 147.

[27] G. Strydom. Xenon-induced Axial Power Oscillations in the 400 MW PBMR. *Nuclear Engineering and Design*, 238:2960–2975, 2008.

[28] S. Trikha and I. Goyal. Diffusion Parameters of Graphite. *Journal of Nuclear Energy Parts A/B*, 20:123–128, 1966.

[29] B. Tyobeka. *Advanced Multi-Dimensional Deterministic Transport Computational Capability for Safety Analysis of Pebble-Bed Reactors.* PhD thesis, Pennsylvania State University, August 2007.

[30] J. Xingqing and Y. Yongwei. Physical Designs and Calculations for the First Full Power Operation of the 10MW High Temperature Gas-Cooled Reactor–Test Module (HTR-10). Institute of Nuclear Energy Technology, September 2004. 2nd International Topical Meeting on HIGH TEMPERATURE REACTOR TECHNOLOGY.

[31] Y. Xu and T. Downar. *GenPMAXS Code for Generating the PARCS Cross Section Interface File PMAXS.* Purdue University School of Nuclear Engineering, November 2006. `https://engineering.purdue.edu/PARCS/Code/Manual/GENPMAXS/PDF/GenPMAXS_nov28_06.pdf`.

[32] M. Young. MELCOR Development for HTGR Applications. From CSARP meeting in Bethesda, MD (ML091060125) `www.getbookee.com/csarp/`, September 2008.